ISSN 0280-5316
ISRN LUTFD2/TFRT--5898--SE

Automatic Implementation and Analysis for Fixed-point Controllers in Modelica using Dymola

Ulf Nordström

Lund University
Department of Automatic Control
May 2012

Lund University **Department of Automatic Control** **Box 118** **SE-221 00 Lund Sweden**	*Document name* MASTER THESIS *Date of issue* May 2012 *Document Number* ISRN LUTFD2/TFRT--5898--SE
Author(s) Ulf Nordström	*Supervisor* Dan Eriksson, Dassault Systemes AB, Sweden Anders Rantzer, Department of Automatic Control, Lund University, Sweden (Examiner) *Sponsoring organization*

Title and subtitle

Automatic Implementation and Analysis for Fixed-point Controllers in Modelica using Dymola (Automatisk implementering och analys av Modelica-baserade fixpunktsregulatorer i Dymola)

Abstract

In model-based development, floating-point arithmetic is typically used for computations in algorithms and models. However, on many target platforms, there is no support for floating-point computations due to constraints on, e.g., price, size, power consumption, and execution speed. A solution is to use fixed-point arithmetic instead. Manually transforming floating-point code to fixed-point code is an error-prone and time-consuming task. Therefore, this thesis has explored the possibility to automatically generate fixed-point code for the controller part of a Modelica system model using Dymola. To support this scenario, Modelica was extended with new experimental annotations and Dymola was extended with additional functionality for analysis and code generation. A complete Modelica model-toembedded code scenario was evaluated using Lego Mindstorms (NXT) as target platform and a tool chain was developed to be able to compile and run the generated code. A Modelica library with components corresponding to the NXT sensors and actuators was also developed. The work has, in addition to this thesis, resulted in two conference papers and has been used in teaching a project course (FRT090) at the Department of Automatic Control at Lund University. It was also used in another Master Thesis where the students generated a stabilizing controller for a two wheeled robot capable of carrying a human.

Keywords

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title 0280-5316		*ISBN*
Language English	*Number of pages* *1-66*	*Recipient's notes*
Security classification		

http://www.control.lth.se/publications/

Preface

This thesis is a part of the degree Master of Science in Electrical Engineering at Lund Institute of Technology.

The work has in addition to this thesis report resulted in two conference papers; main author in *Automatic Fixed-point Code Generation for Modelica using Dymola* and co-author in *Dymola and Modelica_EmbeddedSystems in Teaching – Experiences from a Project Course* in 2009, published in the conference proceeding of the Modelica Conferences 2006 and 2009 respectively.

The work on this project started in October 2005 and has taken many interesting directions since then.

Acknowledgements

I would like to start by thanking my family, especially Anna and Alice, for their never-ending love and support.

I would like to thank everyone at Dassault Systèmes, Lund for their support, especially Hilding Elmqvist, Hans Olsson, Dag Brück, Sven-Erik Mattson, Peter Nilsson and Dan Henriksson who have been supporting me with help and ideas, José Dias Lopes who was my supervisior at Dynasim when the project started and Anders Rantzer, my supervisor at the Department of Automatic Control at LTH.

I would like to thank Johan Åkesson, main-author of one of the conference papers, and course coordinator of FRT090 Projects in Automatic Control. I would like to thank the students who selected the Dymola project in the LTH course FRT090 Projects in Automatic Control given in the spring of 2009, 2010, 2011 and 2012 as well as Andrés and Carlos who used this in their Master's thesis, for their feedback, bug reports and suggestion for improvements as well as patience for using and testing my code in beta status.

Contents

1. Introduction

1.1 Motivation

Hardware-In-the-Loop Simulations (HILS) and Rapid Controller Prototyping (RPC) are widely used today for design and testing of control systems in industry. Typical hardware devices for such tests are Digital Signal Processors (DSP) and Field Programmable Gate Arrays (FPGA). The development of algorithms and models is typically done in high level languages, not directly related to the target hardware. This can be advantageous since conceptual studies can be performed and tested early in the development phase and it also helps to keep the models independent. However, after initial development and studies, code generation for specific target platforms, such as DSPs or FPGAs, has to be done to study the process with the real hardware.

During the development phase, floating-point arithmetics is often used for computations in algorithms and models. However in many applications, economical and technical constraints like price per unit, characteristics of the system and performance of the target platform do not justify the use of such demanding floating-point calculations. Sometimes they are even an obstacle to HILS and production code, since floating-point computations can be to slow for systems with high sampling rates.Integer arithmetic operations execute faster than their corresponding floating-point operations because of their simplicity. In the case of FPGA targets, silicon surface area and power consumption can also be significantly reduced using integer arithmetics. Also, DSP devices often come with only simple arithmetic logic units (ALU), completely lacking hardware support for floating-point arithmetics. Using fixed-point arithmetics, one can utilize the advantages of integer arithmetic operations and generate code for various hardware targets. The achievable precision using integer arithmetics is closely related to the architectural word length of the target platform, typically 16, 24 or 32 bits.

1.2 Problem definition

This thesis explores the possibility to use fixed-point arithmetics for simulation and code generation from models defined using the Modelica language.

Manually transforming equations to fixed-point is a tedious and error prone task. The aim of this thesis is to automatically find a fixed-point mapping of the controller part of a system model to investigate the effects of using finite word length and also to support HILS and RPC by automatically generating fixed-point C code.

Initially, the plan for this Master thesis project was to investigate and implement a way to generate fixed-point code for Modelica models using Modelica code. That work resulted in a conference paper presented at the Modelica Conference in Vienna, Austria, 2006 [1]. That approach turned out to be very hard to further develop and maintain, and furthermore, it was not very user-friendly.

With the later development and official specification of the Modelica language constructs for embedded systems, came the framework that made the current strategy and implementation possible. Using the extensions one can, in a very natural way, partition the system model in different parts, e.g. controller and plant, and map the various parts to different targets, tasks and subtasks. The focus of this work has lead to extend with new functionality to generate fixed-point code for certain target configurations.

1.3 Goals

The main goals of this thesis have been to:

- Extend the Modelica language with experimental language constructs to support specification of properties needed for fixed-point.

- Implement functionality for fixed-point code generation in Dymola based on the experimental language constructs.

- Evaluate fixed-point code generation in a complete Modelica-model to embedded code scenario.

2. Background

This section will to give a short introduction to Modelica and Dymola as well as introducing some basic concepts and notations for fixed-point. The description is intended to be of a general nature to provide the reader with background information and references for further reading.

2.1 Modelica

Modelica

Modelica is a flexible and object-oriented modeling language in the fast growing area of system modeling. In order to use Modelica and the Modelica libraries, a tool with a translator and a compiler is needed in order to run simulations. Dymola [2] is such a tool equipped with a symbolic engine for translation and manipulation of the Modelica code. Dymola also provides features for pre processing of model data and post-processing of simulation data as well as plotting.

The Modelica language is designed for modeling of large, complex, and heterogeneous physical systems. It is a multi-domain language allowing users to combine components from many different engineering domains, such as electronics, mechanics, hydraulics etc. A variety of components from different engineering domains can be seen in Figure 1.

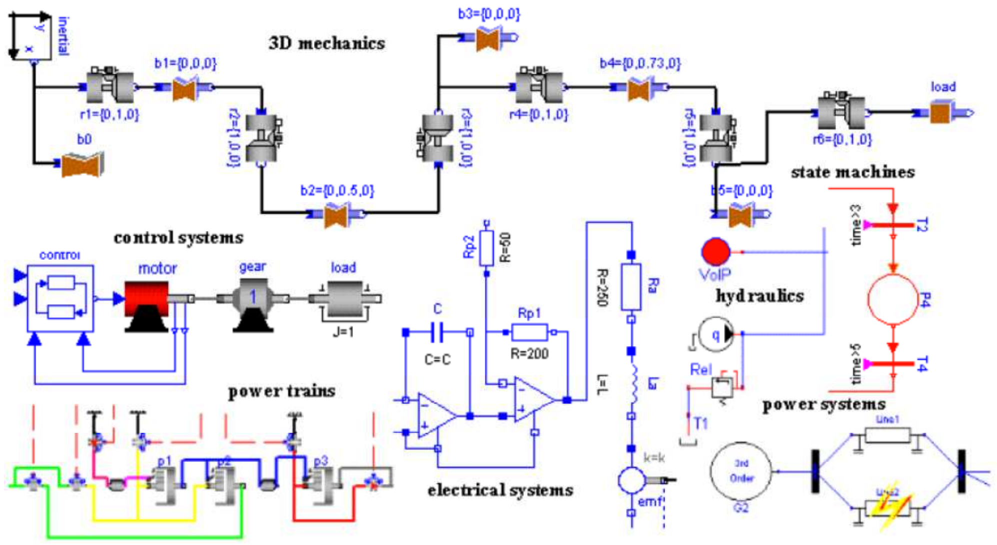


Figure 1 Multi-domain Modelica models.

The design of Modelica allows users to utilize standard components from different, free or commercial, Modelica libraries and, if needed, modify/extend them or design custom components and libraries. An introduction to modeling with Modelica can be found in [3] and [4]. More information regarding Modelica, such as publications, libraries and events can be found in [5].

Dymola

Dymola, *DYnamic MOdeling LAboratory*, is the tool used and extended in this thesis for modeling and simulation with Modelica. It offers a graphical modeling environment and an engine for the necessary symbolic manipulation of equations in Modelica to produce executable code.

In Dymola, a Modelica model can be composed by dragging components from the library browser and dropping them in the diagram sheet. Different components and sub-models are connected and can then be simulated after entering proper component-specific parameters. An alternative to the ”drag-and-drop” technique is to use the underlying text layer, manually describing the behavior of the model using *differential*, *discrete* and/or *algebraic equations*. The text layer can also be used to alter or modify standard components to suit a specific application.

A Modelica model can often contain a huge set of equations, and without some form of symbolic pre-processing, this set is not suitable for numerical integration. Simulation tools, thus, have to include a Modelica translator to manipulate the equations before simulation. Dymola solves this issue using advanced symbolic manipulation techniques. In this way, the set of equations is drastically reduced. After the symbolic manipulation, C-code is generated and compiled with numerical routines into an executable for simulation.

For post-processing and analysis, result files from simulations can be processed in Dymola or by other programs supporting .csv or .mat file formats.

Modelica_EmbeddedSystems

Modelica_EmbeddedSystems (version 0.2, 2009-11-20) is a Modelica library currently under development by the Modelica Association targeting modeling and configuration of embedded systems. In this section the basic components relevant for this thesis will be described. For a more in-depth description see [6] and chapter 16 of the Modelica Language Specification Version 3.1, [7].

The Modelica_EmbeddedSystems library structure can be seen below in Figure 2. The components that are most relevant to this work are CommunicateReal and the records that are used to build configuration records:

- Target record
- Task record
- Subtask record

A task is asynchronous with regards to other tasks and it contains one or more subtasks. A subtask is a synchronous set of equations within a task with the same numerical integration and sampling properties. Each record type will be discussed in a separate section below.

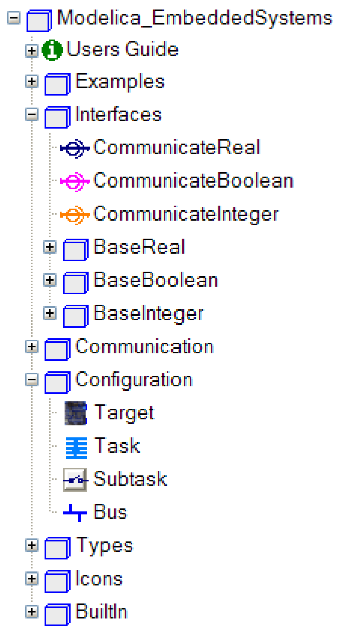
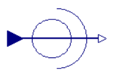
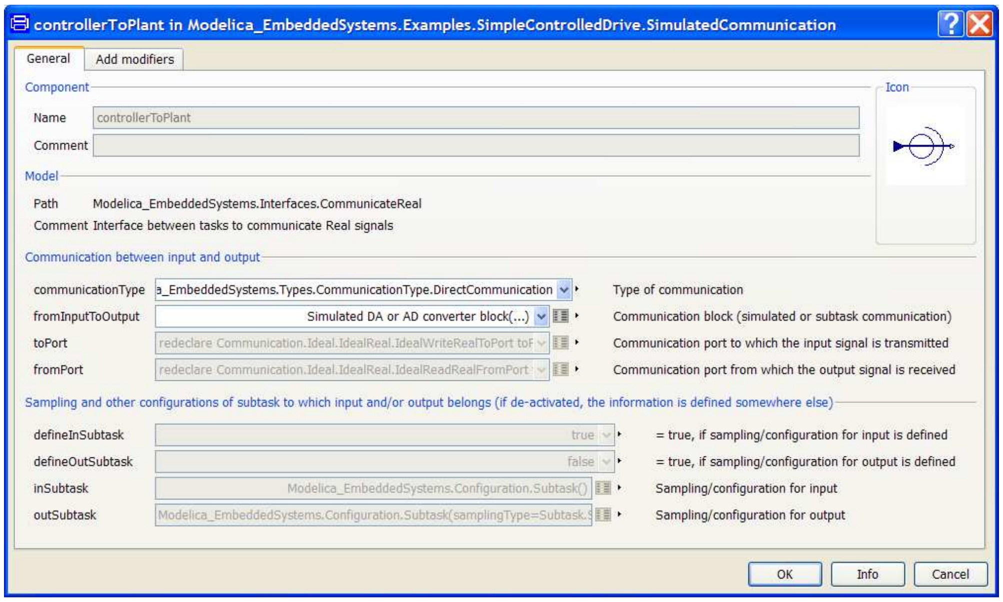


Figure 2 Modelica_EmbeddedSystems library.

Communication points



A communication block (CommunicateReal, CommunicateInteger or CommunicateBoolean) provides a user interface to the underlying Modelica code that is used to decompose the system into tasks and subtasks and define the communication between them as well as to map those on to different targets. The component references a configuration record to collect information of the configuration currently used in the model and use that to “fill” the menus with appropriate choices. This will be demonstrated in chapter 4 when this is used to automatically populate the pull-down menus with Lego Mindstroms API components.



controllerToPlant in Modelica_EmbeddedSystems.Examples.SimpleControlledDrive.SimulatedCommunication

General | Add modifiers

Component

Name: controllerToPlant

Comment:

Model

Path: Modelica_EmbeddedSystems.Interfaces.CommunicateReal

Comment: Interface between tasks to communicate Real signals

Communication between input and output

communicationType	a_EmbeddedSystems.Types.CommunicationType.DirectCommunication	Type of communication
fromInputToOutput	Simulated DA or AD converter block(...)	Communication block (simulated or subtask communication)
toPort	redeclare Communication.Ideal.IdealReal.IdealWriteRealToPort toF	Communication port to which the input signal is transmitted
fromPort	redeclare Communication.Ideal.IdealReal.IdealReadRealFromPort	Communication port from which the output signal is received

Sampling and other configurations of subtask to which input and/or output belongs (if de-activated, the information is defined somewhere else)

defineInSubtask	true	= true, if sampling/configuration for input is defined
defineOutSubtask	false	= true, if sampling/configuration for output is defined
inSubtask	Modelica_EmbeddedSystems.Configuration.Subtask()	Sampling/configuration for input
outSubtask	Modelica_EmbeddedSystems.Configuration.Subtask(samplingType=Subtask.S	Sampling/configuration for output

Figure 3 Parameter dialog of the CommunicateReal component.

In addition to defining the model decomposition, the block is used to define the type of communication between the different tasks/subtasks in the model. The available options that can be selected in the **communicationType** pull-down menu as shown in Figure 3, are:

- Direct communication
- Communication between two subtasks
- Communication between two tasks
- Communication to a port
- Communication from a port

Direct communication is the simplest form of communication ($y = u$) and is used as a starting point when inserting the communication blocks in the model. It basically just propagates the input to the output with the possibility to add noise or delays.

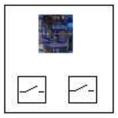
Communication between two subtasks is used to define a border between two subtasks, i.e. the input and the output of the communication block are in different subtasks but belong to the same task.

Communication between two tasks means that the input and output belongs to different tasks. Communication between different tasks is performed in C code external to Modelica.

Communication to a port is used to send information to an I/O port, e.g. sending a signal using Bluetooth via a virtual com port.

Communication from a port is used to receive information from an I/O port.

Configuration records



A configuration record is a Modelica record containing one or more target/task/subtask records to define the configuration of the system. An example, as implemented in the library is depicted below in Figure 4.

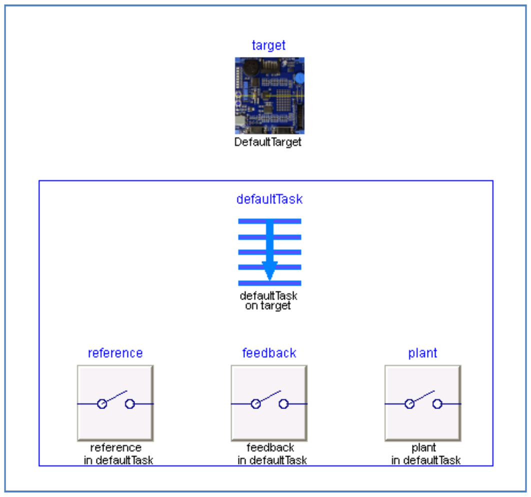
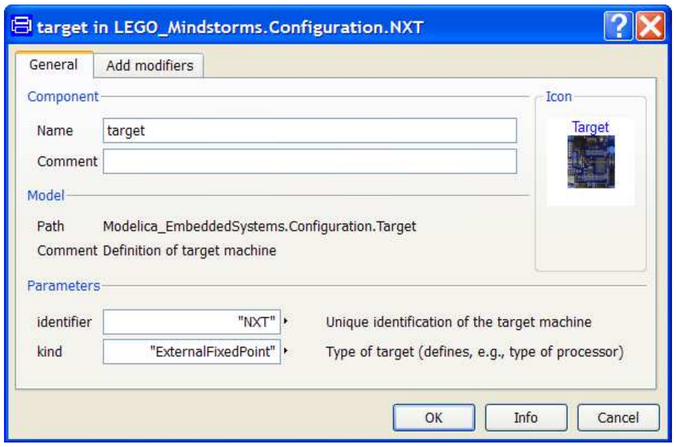


Figure 4 Configuration record from Modelica_EmbeddedSystems.

This particular configuration record has three subtasks that reside in one task on one target. The configuration record is used here to specify different numerical integration methods and sample times for the different parts of the system. In this case, the plant is continuous while the reference and feedback subtasks are periodically sampled and the reference subtask is specified to run five times slower that the feedback subtask. These settings cannot be seen in the image but opening the real model from the library one could pop the parameter dialog and inspect the settings. Other possible configurations could include multiple targets (that will be shown later when discussing investigation of fixed-point arithmetics) and multiple tasks as well.

Target

The Target record contains information on the target. It has two parameters, as shown in Figure 5.


target in LEGO_Mindstorms.Configuration.NXT

General | Add modifiers

Component

Name: target

Comment:

Model

Path: Modelica_EmbeddedSystems.Configuration.Target

Comment: Definition of target machine

Parameters

identifier	"NXT"	Unique identification of the target machine
kind	"ExternalFixedPoint"	Type of target (defines, e.g., type of processor)

Figure 5 Parameter dialog of the Target record.

The **identifier** parameter is used in this thesis to indicate if the target is the host computer CPU,

$$\text{identifier} = \begin{cases} \text{defaultTarget} \\ \text{dymosim} \end{cases}$$

, or the Lego device,

$$\text{identifier} = \text{NXT},$$

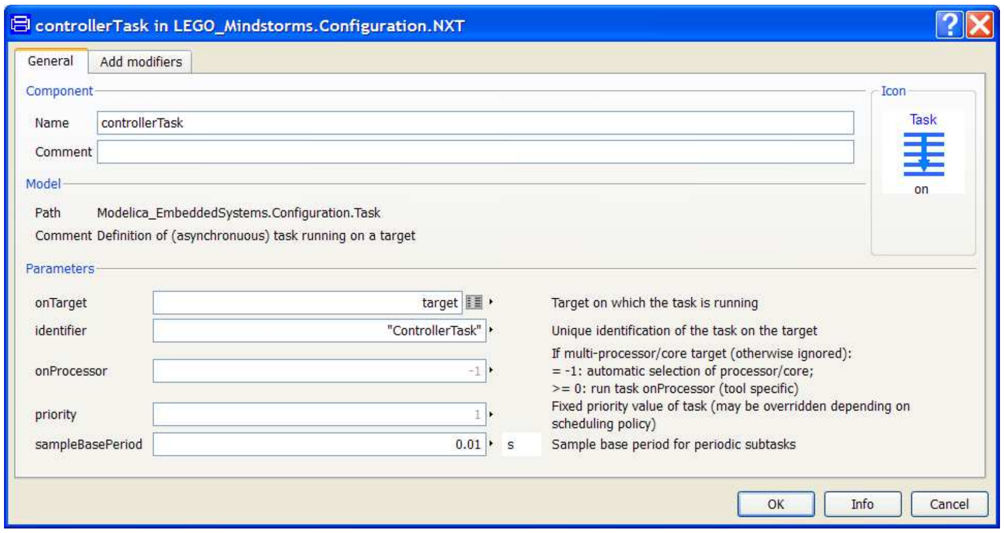
and any other value is interpreted as another external target.

The **kind** parameter is used in this case to indicate that the target does not have a floating-point arithmetic unit and fixed-point code is to be generated for the equations belonging to any task/subtask on that target. To activate set

$$\text{kind} = \begin{cases} \text{InternalFixedPoint} \\ \text{ExternalFixedPoint} \end{cases}$$

Task

The Task record is used as a container for subtasks that are computationally related to each other. The parameter dialog can be seen in Figure 6.


controllerTask in LEGO_Mindstorms.Configuration.NXT

General | Add modifiers

Component

Name: controllerTask

Comment:

Model

Path: Modelica_EmbeddedSystems.Configuration.Task

Comment: Definition of (asynchronous) task running on a target

Icon: Task / on

Parameters

onTarget	target		Target on which the task is running
identifier	"ControllerTask"		Unique identification of the task on the target
onProcessor	-1		If multi-processor/core target (otherwise ignored): = -1: automatic selection of processor/core; >= 0: run task onProcessor (tool specific)
priority	1		Fixed priority value of task (may be overridden depending on scheduling policy)
sampleBasePeriod	0.01	s	Sample base period for periodic subtasks

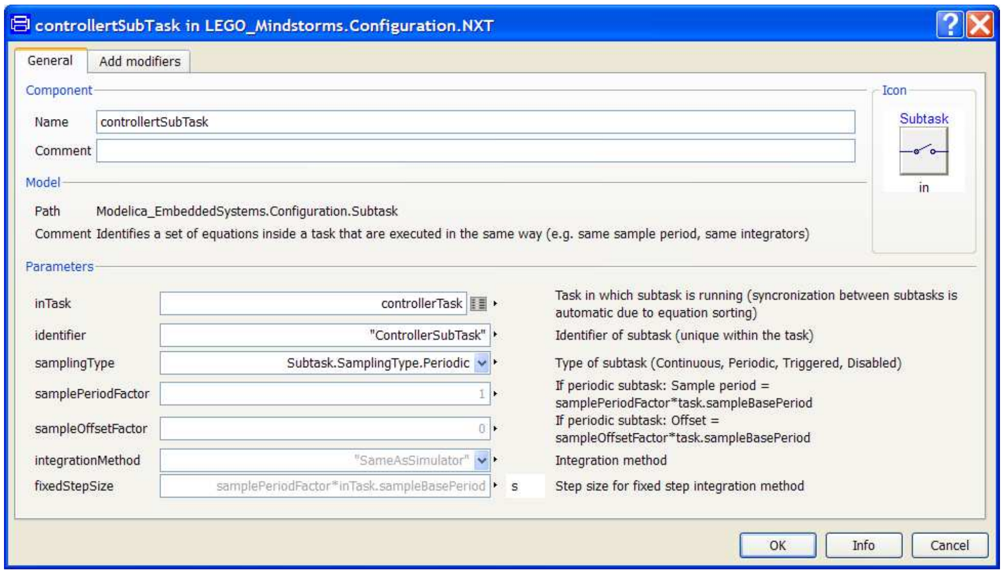
OK | Info | Cancel

Figure 6 Parameter dialog of the Task record.

In this thesis only the **sampleBasePeriod** parameter is of interest, the **identifier** and **onTarget** are used as well, to make the model easier to understand (using good naming) and to specify which target the task runs on. The sampleBasePeriod is used to set the base sample period for periodically sampled subtasks in this task. The subtasks can then be sampled at any integer multiple of the sampleBasePeriod but that configuration is made in the subtask record itself.

Subtask

The subtask record is used to describe sampling properties and numerical integration methods of the subtasks. The parameter dialog is shown in Figure 7.


controllertSubTask in LEGO_Mindstorms.Configuration.NXT

General | Add modifiers

Component

Name: controllertSubTask

Comment:

Model

Path: Modelica_EmbeddedSystems.Configuration.Subtask

Comment: Identifies a set of equations inside a task that are executed in the same way (e.g. same sample period, same integrators)

Icon: Subtask / in

Parameters

inTask	controllerTask		Task in which subtask is running (syncronization between subtasks is automatic due to equation sorting)
identifier	"ControllerSubTask"		Identifier of subtask (unique within the task)
samplingType	Subtask.SamplingType.Periodic		Type of subtask (Continuous, Periodic, Triggered, Disabled)
samplePeriodFactor	1		If periodic subtask: Sample period = samplePeriodFactor*task.sampleBasePeriod
sampleOffsetFactor	0		If periodic subtask: Offset = sampleOffsetFactor*task.sampleBasePeriod
integrationMethod	"SameAsSimulator"		Integration method
fixedStepSize	samplePeriodFactor*inTask.sampleBasePeriod	s	Step size for fixed step integration method

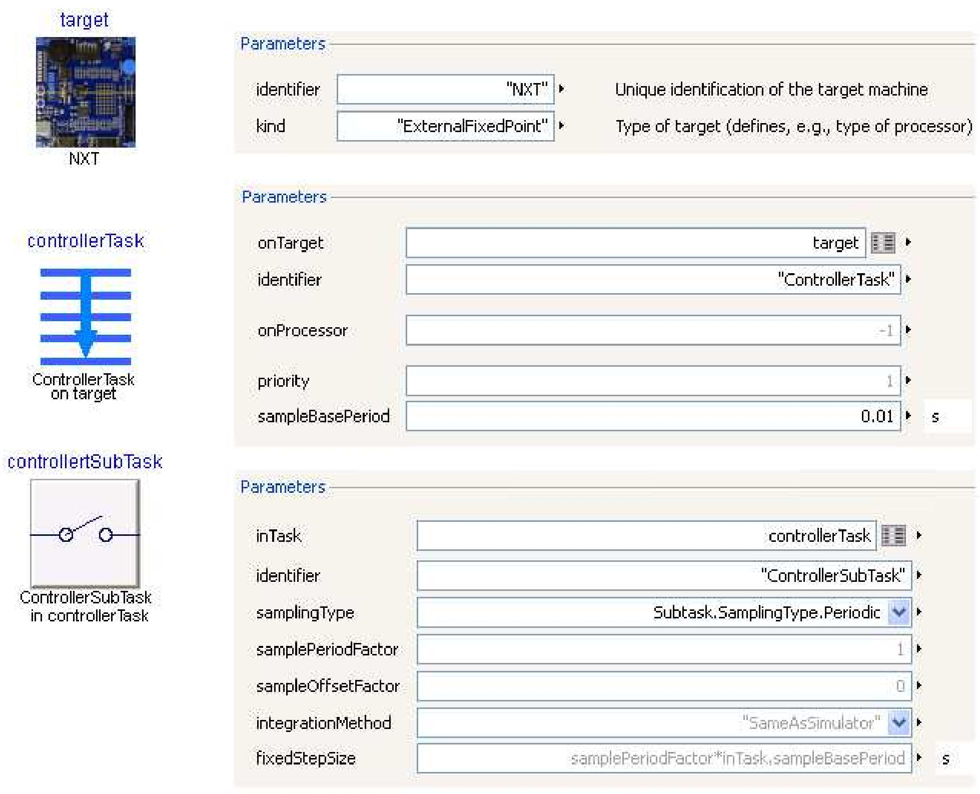
OK | Info | Cancel

Figure 7 Parameter dialog of the Subtask record.

The most important parameters for this thesis are the samplingType and samplePeriodFactor that are used to activate periodic sampling and changing the effective sample period respectively.

Example of Configuration

An example configuration for running the Lego Mindstorms robot with the relevant parts of the parameter dialogs expanded is shown below in Figure 8.



Parameters

identifier	"NXT"	Unique identification of the target machine
kind	"ExternalFixedPoint"	Type of target (defines, e.g., type of processor)

Parameters

onTarget	target	
identifier	"ControllerTask"	
onProcessor	-1	
priority	1	
sampleBasePeriod	0.01	s

Parameters

inTask	controllerTask	
identifier	"ControllerSubTask"	
samplingType	Subtask.SamplingType.Periodic	
samplePeriodFactor	1	
sampleOffsetFactor	0	
integrationMethod	"SameAsSimulator"	
fixedStepSize	samplePeriodFactor*inTask.sampleBasePeriod	s

Figure 8 Exmple configuration for Lego Mindstorms robot.

CommunicationMSWindows

The CommunicateMSWindows library is an add-on library to the Modelica_EmbeddedSystems library developed at DLR (German Aerospace Center, Institute for Robotics and Mechatronics). It contains blocks to access I/O components on a Windows computer, like keyboard, speakers and game controllers. The library structure can be seen below in Figure 9.

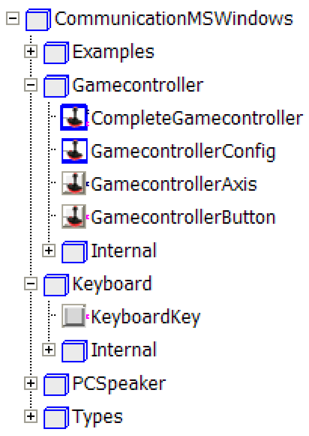


Figure 9 CommunicateMSWindows library from DLR.

In this thesis this library is used to build components that can read signals from a game controller that then can be used for reference signal generation, e.g. driving and steering the Lego Mindstorms robot. A component for the Microsoft SideWinder (steering wheel and throttle/brake pedal) was also implemented and can be used to generate reference signals. Below in Figure 10 is the game controllers that were used.

Figure 10 Logitech Game controller and Microsoft SideWinder.

2.2 Fixed-point

Internally, computers treat and store information using bits, $b \in \{0,1\}$ denoted $\mathbb{Z}_2$. The information in a set of bits has no inherent meaning, it depends entirely on how the data is interpreted. One natural interpretation of bits is as positive integers, coded in natural binary code (NBC), but it is not the only one.

Consider a data byte d represented by 8 bits

$$d = \{b_7, b_6, b_5, b_4, b_3, b_2, b_1, b_0\}, \qquad \forall b_j \in \mathbb{Z}_2.$$

Interpreting the information stored in that byte as a positive integer in NBC, its real world value y represented by d would be

$$y = \sum_{j=0}^{7} b_j \cdot 2^j.$$

The byte $d = 10011101$ would then be interpreted as $y = 157$ since

$$\begin{aligned}
y &= \sum_{j=0}^{7} b_j \cdot 2^j \\
&= 1 \cdot 2^7 + 0 \cdot 2^6 + 0 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 \\
&= 128 + 0 + 0 + 16 + 8 + 4 + 0 + 1 \\
&= 157.
\end{aligned}$$

Another way to interpret the information in d is to treat, for example, $b_4 \cdots b_7$ as an integer and the rest as the fractional part. We then have

$$y = \sum_{j=0}^{7} b_j \cdot 2^{j-4}.$$

and, again with $d = 10011101$, the interpretation would be $y = 9.8125$ since

$$\begin{aligned}
y &= \sum_{j=0}^{7} b_j \cdot 2^{j-4} \\
&= 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 + 1 \cdot 2^{-1} + 1 \cdot 2^{-2} + 0 \cdot 2^{-3} + 1 \cdot 2^{-4} \\
&= 8 + 0 + 0 + 1 + 0.5 + 0.25 + 0 + 0.0625 \\
&= 9.8125.
\end{aligned}$$

which is $157/2^4$. Thus, d can here be used to store both an integer and a decimal value depending on how we interpret the information.

Depending on the interpretation, the information could have virtually any meaning. Integer and decimal values are just examples. The information could also be interpreted as CPU instructions, memory addresses, characters etc.

Data representation

Computers usually use a floating-point representation of real numbers for computations. The floating-point representation allows for numbers in a large span with high resolution. However, when using hardware such as a DSP-processor or an FPGA, the floating-point representation is often not available.

A hardware implementation of floating-point operations like addition and multiplication is very surface- and time-expensive compared to integer operations. Using a fixed-point representation, one can usually achieve faster execution times and more efficient use of the silicon surface area at the cost of reduced precision or limited signal range.

The choice of representation, floating-point or fixed-point, is a tradeoff between precision/range constraints and surface/time constraints. For computations demanding high accuracy in the results, a floating-point representation might be suitable, but for high speed HILS demanding very fast computations, a fixed-point representation that trades reduced precision for speed might be more suitable.

Fixed-point representation

From a hardware point-of-view, fixed-point arithmetics is essentially integer arithmetics with bit shifting. Using integers to represent non-integer values is done by considering an imaginary binary point as follows.

Consider the binary representation of an integer in NBC

$$(b_N, b_{N-1}, \cdots, b_2, b_1, b_0) = \sum_{j=0}^{N} b_j \cdot 2^j, \ \ \forall b \in \mathbb{Z}_2.$$

b_N	b_{N-1}	$\cdots$	b_4	b_3	b_2	b_1	b_0

Figure 11 Binary data representation

Now, using the same set of bits to represent a non-integer value can be done by placing a binary point between $n-1$ and n. Thus

$$(b_N, \cdots, b_{n+1}, b_n, b_{n-1}, \cdots, b_0) = \sum_{j=0}^{N} b_j \cdot 2^{j-n}, \ \ \forall b \in \mathbb{Z}_2 \qquad (2.1)$$

b_N	b_{N-1}	$\cdots$	b_n	b_{n-1}	$\cdots$	b_1	b_0

$\uparrow$

Figure 12 Fixed-point data representation with binary point

The bit to the far most left, b_N, is denoted Most Significant Bit (MSB) and correspondingly we have Least Significant Bit (LSB) to the right.

Range

The integer data type is limited in size by hardware constraints which are machine dependent. For standard CPUs however, an integer q is bounded by

$$0 \le q \le 2^{WL} - 1$$

if unsigned and

$$-2^{WL-1} \le q \le 2^{WL-1} - 1$$

if signed and using two´s complement, see eg. [8], where WL is the word length. Typically we have $WL = 32$ in most modern PC´s but other values are possible, e.g. 16, 24 and 64.

For a fixed-point representation with the binary point at n, the remaining $WL - n$ bits of the word are used to store the integer part and the sign. We thus have the following range for a real variable in fixed-point representation

$$-2^{WL-n-1} \leq y \leq 2^{WL-n-1} - 2^{n}.$$

Resolution

A fixed-point representation with the binary point between $n-1$ and n is said to have n bits of precision. The smallest number that can be represented with that representation is the resolution ε given by

$$\varepsilon = 2^{-n}.$$

With a fixed-point representation it is a trade-off to cover either a large signal range with low precision or a small range with high precision. The combination of large signal ranges and need for high precision leads to a representation using very large WL and most of the benefits of fixed-point can be lost.

So, for signals with large dynamic range and high precision requirements a floating-point representation is usually better. For signals, or equations, where one can accept either limitation in precision or in range, a fixed-point representation can be accurate enough and even increase some critical performance requirements, such as execution time or minimize silicon surface area when implemented in hardware.

Q-notation

The Q-notation is a convenient way to specify a fixed-point representation. It was introduced by Texas Instruments, see e.g. [9].In Q-notation two integers are used to specify the number of bits needed to represent the integer and fractional part of a real number, denoted

$$Q[m, n]$$

or originally

$$Qm.n$$

The integer m is used to represent the number of bits needed for the two´s complement of the integer part and n the number of bits needed for the fractional part. Mapping the Q-notation to the range (for a real value) and resolution described in the section above, the resolution of $Q[m, n]$ is 2^{-n} and the range is

$$-2^{m} \leq y \leq 2^{m} - 2^{n}$$

The total number of bits needed are $WL = m + n + 1$, the extra bit needed to store the sign of the number (as mentioned in the previous section).

3. Implementation and Analysis

This section describes the implementation of fixed-point support in Dymola. It covers the implementation of arithmetic, Boolean, and relational operations, and a couple of basic methods for range analysis. Experimental Modelica annotations used in this report is presented and the structure and details of the generated code is described.

3.1 Fixed-point arithmetics

Conversion

Converting a floating-point value to a fixed-point value is, if given a representation, relatively simple. The task is to find an integer to store the floating-point value in and to find a rule that can be used to recover the floating-point value without losing too much information. Using binary point-only scaling (BPO), this is done by introducing an imaginary binary point as in (2.1). Mathematically, a fixed-point representation q of a floating-point variable y can be described by

$$q = \lfloor 2^n \cdot y \rfloor, \qquad n, q \in \mathbb{Z}, \qquad y \in \mathbb{R}, \tag{3.1}$$

where n is the precision, or equivalently the placement of the binary point left of the LSB, and $\lfloor \cdot \rfloor$ denotes the floor function (other rounding functions could be used to customize the rounding, eg. round towards zero or ceiling). The precision n can be both positive and negative and can be interpreted as a scaling factor, as in (3.1). To recover the floating-point value of a fixed-point representation we just divide with the scaling factor. Hence the recovered value $\tilde{y}$ of a fixed-point representation q is

$$\tilde{y} = \frac{q}{2^n} = 2^{-n} \cdot q, \qquad n, q \in \mathbb{Z}, \qquad \tilde{y} \in \mathbb{R}. \tag{3.2}$$

As an example, consider converting a non-integer value to fixed-point using e.g. $n = 10$ bits of precision. Let us assume that the value to convert is $y = 1.1$. The fixed-point value is then, by (3.1)

$$q = \lfloor 2^{10} \cdot 1.1 \rfloor = \lfloor 1126.4 \rfloor = 1126 \tag{3.3}$$

and the recovered value is, by (3.2),

$$\tilde{y} = \frac{1126}{2^{10}} = 2^{-10} \cdot 1126 \approx 1.099609375. \tag{3.4}$$

It is clear that an error has been introduced by the conversion and recovery since $\tilde{y} \neq y$. In fact, the error comes from the rounding towards zero done by the floor function when converting to fixed-point, the recovery itself is error less. Using (3.1) and (3.2) we can derive a bound on the error by concluding that

$$\lfloor 2^n \cdot y \rfloor \leq 2^n \cdot y < \lfloor 2^n \cdot y \rfloor + 1 \quad \rightarrow$$

$$q \leq 2^n \cdot y < q + 1 \quad \rightarrow$$

$$0 \leq 2^n \cdot y - q < 1 \quad \rightarrow$$

$$0 \leq 2^n \cdot y - 2^n \cdot \tilde{y} < 1 \quad \rightarrow$$

$$0 \leq y - \tilde{y} < 2^{-n}.$$

Hence the maximum magnitude of the error is

$$\sup|\mathrm{y} - \tilde{\mathrm{y}}| = 2^{-\mathrm{n}}.$$

Using the numerical values from (3.3) and (3.4), we have

$$y - \tilde{y} = 0.000390625 < 2^{-10} = 0.0009765625.$$

The smallest number that can be represented is the same as the resolution, 2^{-n}.

In order to assert that no overflow or wraparound occurs

$$\max(|q|) \leq 2^{WL-1} - 1, \qquad \forall q$$

must always hold. This implies that for a given word length there is a limit on the achievable precision. This is closely coupled to the range of the variable since, using the Q-notation described in section 2.2, $WL = m + n + 1$.

Arithmetic operations

The basic arithmetic operations on fixed-point numbers, addition, subtraction, multiplication, and division, are operations with two inputs (the operands) and one output (the result). These operators are usually denoted binary operators. They can be implemented using ordinary integer arithmetic operations and bit shifting. The bit shifts (left shift and right shift) of an integer number q are

$$(q \ll n) = q \cdot 2^n \tag{3.5}$$

and

$$(q \gg n) = \lfloor q \cdot 2^{-n} \rfloor. \tag{3.6}$$

Note that in (3.6) only the integer part of the result is kept and the remainder is discarded. This means that we lose information and errors are introduced. Furthermore, the operation (in e.g. C) is compiler dependent for signed integers taking a negative value [10], so care has to be taken choosing a compiler that interpret right shift as defined above, otherwise obscure and hard-to-trace errors may be introduced. Bit shifting is a fast operation that is used extensively to rescale both the inputs of an operation and the output.

An additional note is that since the implementation of right shift is compiler dependent one can generate code for an integer division with the corresponding power of two instead and let the compiler optimize the code.

Addition

Addition of two fixed-point variables Q_1 and Q_2 on the form $Q[m_1, n_1]$ and $Q[m_2, n_2]$ can be described by finding Q such that

$$Q = Q_1 + Q_2$$

In order to add Q_1 and Q_2, the binary points must be aligned, see. e.g. [11]. This can be done if both Q_1 and Q_2 have the same number of fractional bits, $n_1 = n_2$. This lets us divide fixed-point addition in two different cases;

- Aligned binary points, $n_1 = n_2$
- Unaligned binary points, $n_1 \neq n_2$

If the binary-points are aligned, the two variables can be added, assuming that the result is not larger than the representation can handle. If the binary points are not aligned, then one or more of the operands must be shifted before the addition can be performed. The different cases are discussed in more detail below.

Case $n_1 = n_2$

When $n_1 = n_2 = n$ the two variables can be added according to

$$Q = Q_1 + Q_2$$

and Q will have the same number of fractional bits as the operands, n. The integer part of Q can be stored using $\max(m_1, m_2) + 1$ bits. As a motivating example consider the "worst case" when $Q_1 = Q_2$. Then

$$Q_1 + Q_2 = Q_1 + Q_1 = 2 \cdot Q_1 = Q_1 \cdot 2^1 = Q_1 \ll 1$$

or in words, multiplying with 2 is the same as shifting left with one bit, thus one more bit is needed. Since one more bit is potentially needed there is risk of overflow. Using the Q-notation this could be written as

$$Q[m_1, n] + Q[m_2, n] = Q[\max(m_1, m_2) + 1, n] \tag{3.7}$$

Note that (as explained before) the word length needed to store $Q[m, n]$ is $m + n + 1$ or for the example above

$$(\max(m_1, m_2) + 1) + n + 1 = \max(m_1, m_2) + n + 2 \tag{3.8}$$

Case $n_1 \neq n_2$

If $n_1 \neq n_2$ then the binary points must be aligned before the addition. Essentially this corresponds to shifting one or both operands using left and/or right shifts as previously described. There are several possible shifts that can be performed in order to align the binary points and in order to make a decision a basic strategy must be defined.

As mentioned in the beginning of this section, left shifts are error less (assuming that they do not introduce overflow) and right shifts can introduce an error. In order to lose as little information as possible it is preferable to have no (or as few as possible) right shifts. The basic strategy is then to only use left shifts whenever possible. A constraint is that left shifting can cause overflow. We consider this a sub-case of $n_1 \neq n_2$.

Sub-case $n_1 \neq n_2$ only left shifts

Left shifts correspond to a multiplication with a power of two of the fixed-point representation, (3.5). Given that the operand after shifting fits in the word length, the operand with the smallest number of fractional bits is shifted to make the alignment.

$n_1 > n_2$

$$Q[m_1, n_1] + (Q[m_2, n_2] \ll (n_1 - n_2)) = Q[\max(m_1, m_2) + 1, n_1]$$

with the constraint on Q_2 that

$$m_2 + n_1 + 1 \leq WL$$

in order to avoid overflow.

Example:
Consider $Q_1[10,15], Q_2[5,10]$ and $WL = 32$.

$$Q_1 + Q_2 \colon Q_1[10,15] + (Q_2[5,10] \ll 5) = Q[10,15] + Q[5,15] = Q[11,15].$$

The constraint on $Q_2$is fulfilled since $5 + 15 + 1 < 32$.
Note that we must also assert that the result can be stored in a register, (3.8), e.g.

$$\max(10,5) + 15 + 2 = 27 < 32.$$

Sub-case $n_1 \neq n_2$ left and right shifts

It is not always possible to only use left shifts. Consider the example below where the binary points cannot be aligned by only shifting n_2 since the constraint will not be fulfilled and an overflow would occur.

Example:
Consider $Q_1[10,18], Q_2[15,5]$ and $WL = 32$. According to the basic strategy we would like to use left shift on n_2, since $n_1 > n_2$. Doing that would result in

$$Q_1 + Q_2\text{:}\ \ Q_1[10{,}18] + (Q_2[15{,}5] \ll 13) = Q[10{,}15] + Q[15{,}18] = Q[16{,}18].$$

The constraint on Q_2 will not be fulfilled since $15 + 18 + 1 = 34 \nleq 32$ and we would risk overflow. Furthermore the result would not be possible to store in a register since it would require $16 + 18 + 1 = 35$ bits. In order to avoid this we must assure that the constraint is fulfilled and we have

$$m_2 + (n_2 \ll x) + 1 \leq 32$$

and the largest left shift x that can be performed is

$$x \leq 32 - 1 - m_2 - n_2$$

which for the example gives $x = 11$. Using that we shift Q_2 and get

$$Q_2[15{,}5] \ll 11 = Q_2[15{,}16].$$

Q_2 now has 16 fractional bits and Q_1 needs to be right shifted in order to align the binary points. We have, for Q_1

$$Q_1[10{,}18] \gg x = Q_1[10{,}16]$$

giving $x = 2$. In order to add Q_1 and Q_2 in this case we had to apply both left and right shifts to the operands. The addition becomes

$$\begin{aligned} Q_1 + Q_2\text{:}\ \ &(Q_1[10{,}18] \gg 2) + (Q_2[15{,}5] \ll 11) = Q[10{,}16] + Q[15{,}16] \\ &= Q[16{,}16] \end{aligned}$$

We need $16 + 16 + 1 = 33$ bits to store the result but we only have 32 available. We thus need to modify the shifts to avoid overflow and it is sufficient to reduce the number of fractional bits to 15 instead of 16 in order to be able to store the result. We have

$$\begin{aligned} Q_1 + Q_2\text{:}\ \ &(Q_1[10{,}18] \gg 3) + (Q_2[15{,}5] \ll 10) = Q[10{,}15] + Q[15{,}15] \\ &= Q[16{,}15]. \end{aligned}$$

Before summarizing we introduce a notation for left and right shifts

$$sh(x) = \begin{cases} \ll\ \ x\,, & if\ x > 0 \\ \gg |x|, & if\ x < 0 \\ \ll\ \ 0\,, & if\ x = 0 \end{cases}$$

We then have the following rules for $Q_1 + Q_2$

$$Q_1[m_1, n_1]\ sh(x_1) + Q_2[m_2, n_2]\ sh(x_2) = Q[m, n]$$

where, if $n_1 > n_2$

$$m_2 + n_1 + 2 \leq WL: \begin{cases} x_1 = 0 \\ x_2 = n_1 - n_2 \end{cases} \rightarrow Q[\max(m_1, m_2) + 1, n_1]$$

$$\text{otherwise: } \begin{cases} x_1 = WL - m_2 - n_1 - 2 \\ x_2 = WL - m_2 - n_2 - 2 \end{cases} \rightarrow Q[\max(m_1, m_2) + 1, WL - m_2 - 2]$$

else ($n_1 \leq n_2$)

$$m_1 + n_2 + 2 \leq WL: \begin{cases} x_1 = n_2 - n_1 \\ x_2 = 0 \end{cases} \rightarrow Q[\max(m_1, m_2) + 1, n_2]$$

$$\text{otherwise: } \begin{cases} x_1 = WL - m_1 - n_1 - 2 \\ x_2 = WL - m_1 - n_2 - 2 \end{cases} \rightarrow Q[\max(m_1, m_2) + 1, WL - m_1 - 2]$$

Subtraction

Subtraction follows the same rules as addition and we have for $Q_1 - Q_2$

$$Q_1[m_1, n_1]\, sh(x_1) - Q_2[m_2, n_2]\, sh(x_2) = Q[m, n]$$

where, if $n_1 > n_2$

$$m_2 + n_1 + 2 \leq WL: \begin{cases} x_1 = 0 \\ x_2 = n_1 - n_2 \end{cases} \rightarrow Q[\max(m_1, m_2) + 1, n_1]$$

$$\text{otherwise: } \begin{cases} x_1 = WL - m_2 - n_1 - 2 \\ x_2 = WL - m_2 - n_2 - 2 \end{cases} \rightarrow Q[\max(m_1, m_2) + 1, WL - m_2 - 2]$$

else ($n_1 \leq n_2$)

$$m_1 + n_2 + 2 \leq WL: \begin{cases} x_1 = n_2 - n_1 \\ x_2 = 0 \end{cases} \rightarrow Q[\max(m_1, m_2) + 1, n_2]$$

$$\text{otherwise: } \begin{cases} x_1 = WL - m_1 - n_1 - 2 \\ x_2 = WL - m_1 - n_2 - 2 \end{cases} \rightarrow Q[\max(m_1, m_2) + 1, WL - m_1 - 2]$$

Multiplication

Multiplication of two fixed-point variables Q_1 and Q_2 on the form $Q_1[m_1, n_1], Q_2[m_2, n_2]$ gives a result on the form $Q[m_1 + m_2 + 1, n_1 + n_2]$, according to e.g. [11]. We split multiplication in two cases; the result can be stored and the result is too big to store.

The first case is trivial, just multiplying the factors and getting a result on the form $Q[m_1 + m_2 + 1, n_1 + n_2]$. This result could then be rescaled if needed to remove a portion of the least significant bits. A common choice is to truncate the result so that is does not have more fractional bits than any of the factors. As before note that we need $m + n + 1$ bits to store $Q[m, n]$ and thus we need $m_1 + m_2 + n_1 + n_2 + 2$ bits to store the multiplication.

Example:
Consider multiplying $Q_1[10,2]$ and $Q_2[3,5]$ on a system with $WL = 32$. We then need

$$10 + 3 + 2 + 5 + 2 = 22$$

bits to store the result that will be on the form $Q[14,7]$. In this case the factors can be multiplied like "regular" integers since the result can be stored without manipulation.

The second case is more troublesome since we cannot perform the multiplication without rescaling the factors a priori. Just multiplying them and then trying to rescale the result would potentially destroy the information (due to overflow or wraparound). Some processors have an intermediate register that can hold $2 \cdot WL$ but that is not the case normally. If such a register is available then the multiplication could be performed and the result could be truncated and then returned causing less loss of information.

Since we cannot always count on the availability of a $2 \cdot WL$ register we shift the factors prior to multiplying them to assure that the result can be stored without overflow. We then have

$$\big(Q_1[m_1, n_1]\ sh(x_1)\big) * \big(Q_2[m_2, n_2]\ sh(x_2)\big).$$

We compute $\delta = WL - 1 - (\underbrace{m_1 + m_2 + 1}_{m} + \underbrace{n_1 + n_2}_{n})$ or in words: what is the difference between the result and what we can store, how much must the factors be shifted prior to multiplication in order to make the result fit in $WL - 1$. We are free to distribute the shifts $sh(\delta)$ on the two factors. There has not been time to investigate if there is a distribution that minimizes the error of this operation so the naïve approach to distribute the shifts equally has been chosen whenever possible. Since the shifts are positive this is only possible when δ is an even number. If δ is uneven the factor with the most fractional bits is shifted more than the other. Thus if

$$n_1 > n_2: \begin{cases} x_1 = \left\lceil \dfrac{\delta}{2} \right\rceil \\ x_2 = \left\lfloor \dfrac{\delta}{2} \right\rfloor \end{cases}$$

and

$$n_1 \leq n_2: \begin{cases} x_1 = \left\lfloor \dfrac{\delta}{2} \right\rfloor \\ x_2 = \left\lceil \dfrac{\delta}{2} \right\rceil \end{cases}$$

The result will be on the form

$$\left(Q_1[m_1,n_1]\ sh(x_1)\right) * \left(Q_2[m_2,n_2]\ sh(x_2)\right) \\ = Q[m_1+m_2+1, n_1+x_1+n_2+x_2]$$

Example:
Consider multiplying $Q_1[10,8]$ and $Q_2[13,5]$ on a system with $WL = 32$. To store the result $10+13+8+5+2 = 37$ bits would be needed. In order not to destroy the information we shift the operand according to the rules above. We have

$$\delta = 32-1-(10+13+1+8+5) = 32-1-37 = -6$$

and should thus shift according to

$$\left(Q_1[10,8]\ sh(-3)\right) * \left(Q_2[13,5]\ sh(-3)\right) \\ = (Q_1[10,8] \gg 3) * (Q_2[13,5] \gg 3) = Q_1[10,5] * Q_2[13,2]$$

to get a result on the form

$$Q[10+13+1,5+2] = Q[24,7]$$

that can be stored using $m+n+1 = 24+7+1 = 32$ bits.

Division

For division only a simple rule was implemented to make sure that the fractional bits are set so the results scale correctly. The number of fractional bits for the result, n, is set to

$$n = n_1 - n_2$$

where n_1 and n_2 are the fractional bits of the nominator and denominator.
Interval analysis was used to determine the range of the result, and based on that allocate integer bits.

This simple rule worked with the applications in this thesis but would need improvement for more complex applications. One natural step would be to
left shift the nominator as much as possible, while avoiding overflow, in order to keep as much precision as possible.

Boolean operations

Boolean operators are considered only for single bit Boolean variables, that is, variables that only can take the value true or false, represented by 1 and 0. Denoting a single bit Boolean variable b we have

$$b \in \mathbb{Z}_2 = \{0,1\}.$$

From Boolean algebra and digital circuit theory we find a number of Boolean operators used to represent logical functions. For our purpose, since we only consider the logical functions available in Modelica, only three of them are currently interesting; And, Or and Not. Boolean variables and operations do not introduce any additional errors in the system. This is because the output of a Boolean operator is uniquely determined by the discrete inputs, which are also errorless.

And

Standard Boolean bitwise And operation defined by

$$And(b_1, b_2) = \begin{cases} 1, & if\ b_1, b_2 = 1 \\ 0, & otherwise \end{cases}$$

Or

Standard Boolean bitwise Or operation defined by

$$Or(b_1, b_1) = \begin{cases} 0, & if b_1, b_2 = 0 \\ 1, & otherwise \end{cases}$$

Not

Bitwise inversion, Not, defined by

$$Not(b) = \begin{cases} 0, & if\ b = 1 \\ 1, & if\ b = 0 \end{cases}$$

Relational operations

Denoted relational operations are operators that take fixed-point variables as inputs and produce a Boolean output. They are typically used as conditions in If-Then-Else like constructs to make decisions.
Here we consider; *IfThenElse*, *Equal*, *NotEqual*, *Less*, *LessEqual*, *Greater* and *GreaterEqual*.

Using the fixed-point variables, q_i, and the fixed-point expressions, $expr_i$,

$$q_i, expr_i \in \mathbb{Z}, \qquad cond \in \mathbb{Z}_2$$

the conditional operations are explained below.

Note that in order to make a meaningful evaluation of a condition, for example if two variables are *Equal*, the variables might need rescaling to have the same number of fractional bits, assuming that they are not boolean variables.

Consider again the *Equal* operator. All variables will be considered to be equal if the difference of the two inputs is smaller than the resolution for the given number of fractional bits. This may cause errors, e.g. selecting the wrong branch in an If-Then-Else construct. This reasoning is valid for all conditional operators considered here.

As always, when shifting fixed-point variables, care needs to be taken so that the shifts do not cause overflows.

If-Then-Else

The *If-Then-Else* operator evaluates a Boolean condition and returns a fixed-point value or expression. It is used in discrete and/or Boolean equations to evaluate different branches depending on the condition. It is a ternary operator, i.e., takes three inputs, and the generic syntax of the *If-Then-Else* statement is

$$IfThenElse(cond, expr_1, expr_2) = \begin{cases} expr_1, & if\ cond = 1 \\ expr_2, & if\ cond = 0 \end{cases}$$

Equal

The *Equal* operator is equivalent to the standard syntax ”==” operator.

$$Equal(q_1, q_2) = \begin{cases} 1, & if\ q_1 = q_2 \\ 0, & otherwise \end{cases}$$

Not equal

The *NotEqual* operator is equivalent to the standard syntax ”! =” operator.

$$NotEqual(q_1, q_2) = \begin{cases} 1, & if\ q_1 \neq q_2 \\ 0, & otherwise \end{cases}$$

Less

The *Less* operator is equivalent to the standard syntax ”<” operator.

$$Less(q_1, q_2) = \begin{cases} 1, & if\ q_1 < q_2 \\ 0, & otherwise \end{cases}$$

Less equal

The *LessEqual* operator is equivalent to the standard syntax ”<=” operator.

$$LessEqual(q_1, q_2) = \begin{cases} 1, & if\ q_1 \leq q_2 \\ 0, & otherwise \end{cases}$$

Greater

The *Greater* operator is equivalent to the standard syntax ”>” operator.

$$Greater(q_1, q_2) = \begin{cases} 1, & if\ q_1 > q_2 \\ 0, & otherwise \end{cases}$$

Greater equal

The *GreaterEqual* operator is equivalent to the standard syntax ”>=” operator.

$$GreaterEqual(q_1, q_2) = \begin{cases} 1, & if\ q_1 \geq q_2 \\ 0, & otherwise \end{cases}$$

Relative and absolute resolution

Resolution, as defined before, is a measure on the smallest number that can be represented with a fixed-point representation using n bits of precision. For variables and parameters it can be given using either relative or absolute resolution.

The relation between relative resolution, ε_{rel}, and absolute resolution, ε_{abs}, for a variable, y, is defined by:

$$\mathcal{R}(y) * \varepsilon_{rel} = \varepsilon_{abs}$$

where $\mathcal{R}(y) = \max(|y_{min}|, |y_{max}|)$ is the range of y.

Typically, when the range of a variable is known the relative resolution is a good way to specify the resolution but in some cases it is very convenient to use absolute resolution instead. An example could e.g. be a signal coming from an AD converter where the number of bits used is known.

Example
A relative resolution of 0.001 in the range 100 is

$$100 * 0.001 = 0.1$$

i.e., the absolute resolution is 0.1.

Using (3.2) we see that this requires four fractional bits

$$2^{-4} < \varepsilon_{abs} < 2^{-3}$$
$$0.0625 < 0.1 < 0.125$$

Example
An AD converter outputs a signal with ten bits. The absolute resolution is

$$\varepsilon_{abs} = 2^{-10} \approx 0.00098$$

As can be seen above the number of fractional bits, nF, is directly related to the resolution. The exact relation can be found by

$$\varepsilon_{abs} = 2^{-nF} = \frac{1}{2^{nF}} = \mathcal{R}(y) * \varepsilon_{rel} \quad \rightarrow$$
$$2^{nF} = \frac{1}{\mathcal{R}(y) * \varepsilon_{rel}} \quad \rightarrow$$

$$nF = \log_2(\frac{1}{\mathcal{R}(y) * \varepsilon_{rel}})$$

Since $nF \in \mathbb{Z}$ it needs to be rounded, and to be able to represent the resolution we require that

$$2^{-nF} \leq \varepsilon_{abs}$$

$$2^{nF} \geq \frac{1}{\mathcal{R}(y) * \varepsilon_{rel}}$$

$$nF \geq \log_2(\frac{1}{\mathcal{R}(y) * \varepsilon_{rel}})$$

which holds for

$$nF = \left\lceil \log_2\left(\frac{1}{\mathcal{R}(y) * \varepsilon_{rel}}\right) \right\rceil$$

In Dymola the resolution is used to compute the number of fractional bits needed. The implementation allows both relative and absolute resolution as input, which will be shown in section 3.3.

Limitations

Many features of the Modelica language are supported, eg. arrays, matrices, while-loops, algorithm sections, etc., but there are some limitations. The most important ones are:

- No function calls (Modelica, built-in nor external)
- No for-loops
- ^ (power operator) not supported

3.2 Range analysis

To achieve high precision in the arithmetic operations it is important to make good use of the available bits. This means to allocate enough integer bits to make sure that there is no overflow while not losing too much precision.

As a motivating example consider an expression on a system with an 8-bit word length

$$q = q_1 + q_2$$

If no information on the range $\mathcal{R}(q) = q_1 + q_2$ of the expression is available, one simple approach is to split the available bits evenly between the integer and fractional part. For this example that would be either $Q[3{,}4]$ or $Q[4{,}3]$ (remembering that $Q[m, n]$ needs $m + n + 1$ to store). This gives the result, q, three or four bits of precision assuming that the inputs, q_1 and q_2, are known with precision higher or equal to that. Let us now consider that it is known that the range of the expression is in the interval $[0{,}1]$. The expression can then be on $Q[1{,}6]$ format and the result would have 6 bits of precision instead of 3 or 4.

Using range analysis the range of all expressions and sub-expressions can be determined more or less accurately depending on what method is used and what data is available on the signals. A couple of different approaches are presented next.

Bit propagation

Bit propagation is a coarse method to approximate the range of an expression and how it grows in the expression tree. The idea is to propagate the fixed-point representation through the expression to get an estimate on the range.

As we know from previous sections, (3.7), adding two fixed-point numbers requires them to have the same number of fractional bits, n. In the example below.

$$[m_1, n] + [m_2, n] = [\max(m_1, m_2) + 1, n]$$

two fixed-point numbers are added and we see that the number of integer bits is increased by one for each addition. The analogy for multiplication is

$$[m_1, n_1] \cdot [m_2, n_2] = [m_1 + m_2 + 1, n_1 + n_2]$$

and we see that the integer bits grow rapidly.

Interval arithmetics

Interval arithmetics can be used to estimate intervals of expressions and intermediate results. It was introduced in the 1960´s by R. E. Moore in [12].Unfortunately, interval arithmetics also often results in an overestimate of the resulting intervals.

The basic propagation rules, see eg. [13], are

$$[a, b] + [c, d] = [a + c, b + d]$$

$$[a, b] - [c, d] = [a - d, b - c]$$

$$[a, b] * [c, d] = [\min\,(ac, ad, bc, bd), \max\,(ac, ad, bc, bd)]$$

$$\frac{[a, b]}{[c, d]} = \left[\min\left(\frac{a}{c}, \frac{a}{d}, \frac{b}{c}, \frac{b}{d}\right), \max\left(\frac{a}{c}, \frac{a}{d}, \frac{b}{c}, \frac{b}{d}\right)\right]$$

where 0 is not allowed to be in $[c, d]$ for division.
These rules have been implemented to support interval arithmetics in Dymola. We also have

$$if\ expr\ then\ [a, b]\ else\ [c, d] = [\min(\mathrm{a}, \mathrm{c}), max(b, d)]$$

for *if-then-else* expressions. For interval arithmetics the ranges of the variables are propagated though the expressions giving resulting intervals for intermediate results. The range of the variables themselves can be input by the user or it can be derived/set by heuristics.

Simulation-based approach

Monte Carlo simulations can be used to cover a wide range of use cases and get a good approximation on the ranges of variables and expressions. Given a set of cases, one can expect good results on the accuracy of the ranges. However, it is time-consuming to perform exhaustive sets of simulations and it is hard, if not impossible, to know if all cases are covered.

There are ways to configure Dymola for Monte Carlo simulations but the infrastructure to collect and push range information from Monte Carlo simulations back into the model was missing and therefore this approach was never evaluated.

Approach in Dymola

The approach used for range analysis in this work is interval arithmetics. An implementation of bit propagation was also tested but discarded since interval arithmetics gave tighter intervals.

3.3 Experimental annotations

In order to be able to specify resolution for a Modelica variable a new experimental annotation has been introduced. It has the syntax

```
annotation(fixedpoint(resolution=<Real value>));
```

to specify the relative resolution, or

```
annotation(fixedpoint(bits=<Integer  value>));
```

to specify an absolute resolution by specifying the (positive or negative integer) number of fractional bits for the variable. A negative number of fractional bits can be view as a scaling, just as a positive number, to reduce the size of large signals.

Example of annotated variables with min and max:

```
Real I(min=-10, max = 10) annotation(fixedpoint(resolution=0.001));

output Real to_DA(min=0, max=1) annotation(fixedpoint(bits=8));
```

The annotations can be used as modifiers (not allowed in Modelica language specification) by enabling special support in Dymola. The reason is that the user should be able to input additional information (like resolution, min and max) in a model just by extending from a base model and modifying it, thus working in a "true" object-oriented style, reusing the base model.

3.4 Defaults and heuristics

In order to get a starting point, to let a user start experimenting without having to set min/max and resolution for all variables, default values for min, max and resolution are used in many cases. These apply to two categories of variables; parameters and variables. Exception are inputs to the system and in many cases discrete state variables. Inputs must be user annotated and if not, the translation will be aborted and an error message will list the variables that must be annotated in order to proceed.

With these requirements and the rules for default values described here, a starting point for fixed-point simulations is set without the need to provide min/max and resolution for all variables. The user can, and should, then modify the defaults to use better (more confident) min/max values, and to use resolution to control what variables need the most resolution in order to keep the system within tolerable limits using fixed-point arithmetics instead of floating-point.

Parameters

Parameters are variables that are constant during simulation. What separates them from constants in Modelica is that the user should have the possibility to change them without recompiling the model. An implication of this is that parameters cannot be converted to fixed-point literals, like constants and real literals, and hard-coded for better performance. The constants, or literals, can be converted once, at translation time, whereas the parameters need to be converted during runtime in the host.

For parameter the following default values are used:

- $min = -2 \cdot |value|$
- $max = 2 \cdot |value|$
- $relative\ resolution = 10^{-4}$

where $value$ is the value assigned to the parameter at translation time. The implementation allows the default value to be overridden by setting the Dymola flag `Advanced.ParameterResolution`.

Variables

A variable that is not an input to the system, a discrete state variable or a parameter can be computed using inputs, states and parameters or other variables that are already known (equation systems are not supported for fixed-point handling in Dymola). This means that the min/max can be derived using interval arithmetics for all variables that are not in the categories mentioned above, with the exception of discrete state variables in some cases, an example will be given below. Hence, the min/max defaults specified below are just symbolic, after translation they are normally overwritten by a derived or a user-specified min/max value.

For variables the following default values are used:

- $min = \begin{cases} -10^{-60}, & if\ real \\ -2147483647, & if\ integer \end{cases}$
- $max = \begin{cases} 10^{-60}, & if\ real \\ 2147483647, & if\ integer \end{cases}$
- $relative\ resolution = 10^{-8}$

As for parameters, the implementation allows the default value to be overridden by setting the Dymola flag `Advanced.VariableResolution`.

As an example when the min/max for a discrete state variable can be derived, consider a PID controller where we have two discrete states; i and e, described by the Modelica block below.

```
block SimplePID
  parameter Real Gain(min=0.1, max=2)=1;
  parameter Real DT(min=0.02, max=1)=0.1;
  parameter Real Ti(min=0.1, max=100)=100;
  parameter Real Td(min=0, max=2)=0;
  input Real Sp(min=-10, max=10) annotation(fixedpoint(resolution=1e-5));
  input Real Pv(min=-10, max=10) annotation(fixedpoint(resolution=1e-5));
  output Real C annotation(mapping(...));
protected
  Real e(start=0, fixed=true);
  Real i(start=0, fixed=true, min=-100, max=100) annotation(fixedpoint(resolution=1e-
5));
equation
  when sample(0,DT) then
    e = Sp - Pv;
    i = pre(i) + e;
    C = Gain*(e + DT/Ti*i + Td/DT*(e-pre(e)));
  end when;
end SimplePID;
```

Using a ramp as set point signal and connecting the controller to a simple process we get a small system

```
model Sys
  SimplePID pid;
  Process proc;
equation
  pid.Sp = if time < 0.5 then 0 else 1;
  pid.Pv = Subtask.decouple(proc.y);
  proc.u = Subtask.decouple(pid.C);
end Sys;
```

`Subtask.decouple()` is a Modelica operator that is used by Dymola to partition the system in to different subtasks, described in [7]. For the system above the following declarations is generated based on the annotations and interval analysis.

```
/* input Real pid.Sp(min = -10.0, max = 10.0)
annotation(fixedpoint(resolution = 1E-005)); */
int_16 pid_Sp_FP = 0;
/* Q[1, 14] Derived: min = 0.0, max = 1.0 */

/* input Real pid.Pv(min = -10.0, max = 10.0)
annotation(fixedpoint(resolution = 1E-005)); */
int_16 pid_Pv_FP = 0;
/* Q[1, 14] Derived: min = -1.0, max = 1.0 */

/* discrete Real pid.e */
int_16 pid_e_FP = 0;
/* Q[2, 14] Derived: min = -1.0, max = 2.0 */

/* discrete Real pid.i(min = -100.0, max = 100.0)
annotation(fixedpoint(resolution = 1E-005)); */
int_32 pid_i_FP = 0;     /* Q[7, 10] */

/* output discrete Real pid.C */
int_32 pid_C_FP = 0;
/* Q[12, 19] Derived: min = -2602.0, max = 2604.0
*/
```

```
/* parameter Real pid.DT(min = 0.02, max = 1.0) = 0.1
*/
int_16 pid_DT_FP = 1638; /* Q[1, 14] */

/* parameter Real pid.Gain(min = 0.1, max = 2.0) = 1 */
int_16 pid_Gain_FP = 8192;    /* Q[2, 13] */

/* parameter Real pid.Td(min = 0.0, max = 2.0) = 0 */
int_16 pid_Td_FP = 0;    /* Q[2, 13] */

/* parameter Real pid.Ti(min = 0.1, max = 100.0) = 100
*/
int_16 pid_Ti_FP = 12800;     /* Q[7, 7] */
```

As can be seen above the min/max of the output, `C`, is derived by interval analysis based on the min/max of other variables. For the input variables, `Sp` and `Pv`, the derived min/max gave a narrower interval than specified by the user. The user specified min/max attributes are then ignored.

Also note the discrete state variable, `e`, which only depends on the inputs and therefore the min/max could be derived.

3.5 Code Generation

Typically, when translating a Modelica model in Dymola, C code for the model equations is generated and compiled into an executable that can be run on the PC. This executable (dymosim.exe) is the actual simulator that integrates/solves the model equations. This “normal” case when simulating a model in Dymola is supported for fixed-point code generation and we denote it *Internal fixed-point* in this report.

Using Modelica_EmbeddedSystems we can, e.g., separate the controller part in a system from the plant and map that to an external target. Fixed-point code generation for this scenario is denoted *External fixed-point*.

The main difference for the scenario above is that the actual computations of the fixed-point task are performed on the external target instead of on the PC. The simulator, dymosim, is in this case only used for (optional) data logging for plotting and animation on the PC.

Internal fixed-point

The ´Internal fixed-point´ mode is typically used to study effects and control performance using fixed-point computations in a controller compared to using floating-point. The effects of fixed-point computations can, e.g., introduce unwanted limit cycles due to quantization and overflows affecting system stability.

For an efficient workflow it is essential to easily be able to compare signals and the effect they have on the system. To make it convenient for users, code is generated such that both fixed-point and floating-point computations are computed. A Boolean variable in the GUI (accessible in the Dymola variable browser) can then be used to select if the result of fixed-point or floating-point computations shall be used as output. The benefit of this approach is that the model only needs to be translated and compiled once. Simulations can then be performed using either fixed-point or floating-point by toggling the GUI switch without re-translation.

Below the code for the actual model equations is described in *Equations* and the declaration of fixed-point variables in *Declarations*. Note that the fixed-point variables/equations are not available in the Modelica code of the model; they are created during translation and realized in the C code next to the original equations.

When in the *Internal fixed-point* mode, all code is generated in the same file where Dymola outputs the normal model equations, called dsmodel.c.

Declarations

The actual fixed-point variables are declared as local integer variables in the C code and are not stored in the result file after a simulation. Example:

```
/* Real y */
int_32 y_FP = 0;
/* Q[11, 11] Derived: min = -2.0, max = 1202.0 */

/* parameter Real q1 = 0.5
annotation(fixedpoint(bits = 11)); */
int_16 q1_FP = 1024;
/* Q[1, 11] Derived: min = -1.0, max = 1.0 */

/* parameter Real q2(min = 0.0, max = 1000.0) =
0.5 annotation(fixedpoint(bits = 4)); */
int_16 q2_FP = 8;   /* Q[10, 4] */

/* parameter Real q3 = 0.5
annotation(fixedpoint(bits = 5)); */
int_8 q3_FP = 16;
/* Q[1, 5] Derived: min = -1.0, max = 1.0 */

/* parameter Real q4(min = 0.0, max = 100.0) = 0.5
annotation(fixedpoint(bits = 10)); */
int_32 q4_FP = 512; /* Q[7, 10] */

/*  time */
int_32 time_FP = 0; /* Q[7, 10] */
```

Instead new variables are created and the rescaled values of the fixed-point variables are mapped back during simulation.

For a variable `y` we declare in dsmodel.c as a local variable

`<int_type>` y_FP

This is the actual integer variable that holds the value of the fixed-point computation. Furthermore, we also introduce two new versions of the Modelica variable under the virtual Modelica component `<fixedpoint>`. These two variables are

```
y_fromfixedpoint
```

which is the recovered value of the fixed-point variable and

```
y_original
```

which is the value of the original equation computed using floating-point arithmetics. This gives a hierarchy in the stored result (that is also reflected in the Dymola variable browser, see e.g. Figure 20). Note that those two variables are not declared as local C variables, they are instead declared as if they existed as real variables in the model. The reason is that it is then possible to get a nice structure as described below.

As an example of the structure consider three variables `a,b,c,` where `b` and `c` belong to the fixed-point task. We then get

```
<a>
<b>
<c>
<fixedpoint>
   | - <b>
   |      | - fromfixedpoint
   |      | - original
   | - <c>
   |      | - fromfixedpoint
   |      | - original
```

Below, in the *Equations* section, the implementation and interpretation of the new variables are described.

Equations

As mentioned before, code is generated such that equations are computed in both floating-point and fixed-point. Below is an example in C code for the variable `y` from the previous example. As can be seen, the two new variables introduced in the previous section, `y_fromfixedpoint` and `y_original` are always computed and then only one is used to assign the original variable depending on the value of the Boolean toggle variable `useFixedPoint`.

```
/* Fixedpoint equations */
/* y = q1+q2+q3+q4+time; */
y_FP = ((((q1_FP + (q2_FP << 7)) + (q3_FP << 6))
 + (q4_FP << 1)) + (time_FP << 1));
/* Mapping from fixedPoint variables to Modelica
 variables */
/* y = 2^(-11)*y.FP */
fp_y_fromfixedpoint = 0.00048828125*y_FP;
fp_y_original = q10_0+q20_0+q30_0+q40_0+Time;
y0_0 = useFixedPoint0_0 ? fp_y_fromfixedpoint :
 fp_y_original;
```

The generated code contains automatically generated comments for improved readability and traceability.

First in the block is a comment with the original Modelica equation followed by the actual code for the corresponding fixed-point variable. Then, before the mapping back to a Modelica variable, the scaling is clarified in a comment.

External fixed-point

The External fixed-point mode is used to generate code that can be incorporated in a framework and downloaded on an external target platform for further testing and verification. To keep the code portable, it is split into two files; declarations.c and equations.c, containing the variable declarations and the fixed-point model equations, respectively. These two files can then be included in a user written framework, see. e.g. *dymola_wrapper* in section 4.3 for an example implementation. Code for data logging and interaction (eg. reference signal generation using a game-pad) is optionally generated for the normal simulator, dymosim, in the file dsmodel.c.

Declarations

The declarations of local fixed-point variables are in declarations.c, which is included by the framework code for the external target. In dsmodel.c, the same file is included if data logging is enabled to get consistent variable declarations. An example is:

```
/* Type definitions for fixedpoint data types */
#ifndef DYMOLA_FP_TYPES
#define DYMOLA_FP_TYPES
  typedef          char bool_8;
  typedef          char int_8;
  typedef     short int int_16;
  typedef           int int_32;
  typedef long long int int_64;
#endif

/* input Modelica.Blocks.Interfaces.RealInput
sendMotorA.u(min = -100.0, max =
  10.0) annotation(fixedpoint(bits = 0)); */
int_8 sendMotorA_u_FP = 0;    /* Q[4, 0] Derived:
min = 0.0 */

/* input Modelica.Blocks.Interfaces.RealInput
sendMotorB.u(min = 0.0, max = 100.0) */
int_16 sendMotorB_u_FP = 0;   /* Q[7, 8] */

/* parameter Real ramp.height = 100 */
int_16 ramp_height_FP = 6400; /* Q[8, 6] Derived:
min = -200, max = 200 */
```

In the example above the type definitions are followed by the declaration of two input variables `sendMotorA_u_FP` and `sendMotorB_u_FP` and a parameter `ramp_height_FP`.

As can be seen, the declaration of a fixed-point variable includes some additional comments to improve the traceability to the original variable in the Modelica model, as well as detailed information on the fixed-point representation of the variable.

First is a comment containing the original declaration as found in the Modelica code. It is followed by the declaration of the fixed-point variable. Note that the fixed-point variable is automatically initialized with the fixed-point converted value of its real Modelica value.

In the example above the fixed-point parameter `ramp_height_FP` is automatically initialized to 6400 which is the value of the Modelica parameter ramp.height=100 scaled up using the Q[8, 6] format.

$$100 * 2^6 = 6400$$

Finally there is a comment with the Q notation of the selected fixed-point representation as well as derived min/max values for those variables where this was not set by the user. The min/max, either derived or original, can be used to verify the integer part of the fixed-point representation.

Equations

The fixed-point model equations are generated in equations.c which can be included in a framework assuming that the declarations.c file already has been included. As can be seen in the example code below, for a Lego Mindstorms NXT target, each fixed-point equation is preceded by a comment containing the original Modelica equation for traceability. The block at the end is optionally generated if data logging is enabled.

```
/* sendMotorA.u = (if time < ramp.duration then
time*ramp.height/ramp.duration
   else ramp.height); */
sendMotorA_u_FP = ((((time_FP < (ramp_duration_FP
<< 4)) ? ((time_FP *
  ramp_height_FP) / ramp_duration_FP) :
(ramp_height_FP << 4)))) / 2048;
/* readMotorA.y =
LEGO_Mindstorms.Communication.ExternalC.ECRobot.Se
rvoMotor.nxt_motor_get_count
  (readMotorA.fromPort.n, time); */
readMotorA_y_FP =
nxt_motor_get_count(readMotorA_fromPort_n_FP);
/* Sending variables using bluetooth */
target_port_bufwrite_int32(sendMotorA_u_FP);
target_port_bufwrite_int32(readMotorA_y_FP);
target_port_write_flush();
```

The corresponding code in dsmodel.c when data logging is enabled doesn't contain any equations from the model. All equations are replaced with a call to read the actual value from the external target for data logging purposes. In the example below a generic read function, `host_port_read_int32()`, is called to read an integer from the target. For this thesis an implementation to read from the Lego Mindstorms NXT device was developed. As can be seen in the example the variable is calculated by reading the value from the target and then it is rescaled to the Modelica variable. As for the Internal fixed-point model the variable is preceded by a comment containing its original equation for traceability.

```
/* sendMotorA.u = (if time < ramp.duration then
time*ramp.height/ramp.duration
     else ramp.height); */
sendMotorA_u_FP = host_port_read_int32();
/* Mapping from fixedPoint variables to Modelica
variables */
/* sendMotorA.u = 2^0*sendMotorA.u.FP */
sendMotorA_u = sendMotorA_u_FP;
/* readMotorA.y =
LEGO_Mindstorms.Communication.ExternalC.ECRobot.Se
rvoMotor.nxt_motor_get_count
    (readMotorA.fromPort.n, time); */
readMotorA_y_FP = host_port_read_int32();
/* Mapping from fixedPoint variables to Modelica
variables */
/* readMotorA.y = 2^0*readMotorA.y.FP */
readMotorA_y = readMotorA_y_FP;
```

If no data logging (and no Bluetooth communication from PC to target) then the target is completely free from the PC and can be run on its own. The example code for the Lego Mindstorms NXT target above would then not contain the section starting with

```
/* Sending variables using bluetooth */
```

3.6 Automatic error-checking code

To help users evaluate the fixed-point representation they are using, code can be generated to automatically check the error. The current implementation is based on generating assert statements that are evaluated after every computation of the variable. This will slow down the code and is not applicable when running on an external target so it is not enabled by default. Furthermore, the current strategy is not sophisticated enough, a small phase shift in the signal can give a large error in magnitude and the simulation will stop.

For the error checking code to be usable, more development is needed, which is outside the scope of this thesis. The main idea with this implementation is to show that it is possible to integrate that type of code generation in the framework automatically.

The current implementation uses the following error criteria for a variable y.

$$assert\left(\left|y_{fromfixedpoint} - y_{original}\right| \leq 0.001 \cdot \left(\left|y_{original}\right| + y_{nominal}\right)\right)$$

where $y_{fromfixedpoint}$ is the recovered value from the fixed-point computations, $y_{original}$ is the original value from the floating-point computation and $y_{nominal}$ is the nominal value of the variable (default value in Dymola is $y_{nominal} = 1$ except for pressure variables where $y_{nominal} = 10^5$).

Checking of min/max attributes set by the user can be done by enabling assertions for min/max in Dymola.

4. Application Examples

To evaluate the code for External fixed-point the Lego Mindstoms NXT device was used as target platform. A Dymola Lego Mindstorms API in the form of a Modelica library was developed to support the platform.

This section describes some application examples, starting with the platform, tool chain and Dymola Lego Mindstorms API. Online plotting over Bluetooth and the Lego Segway are introduced as application examples.

4.1 Platform and tools

Lego Mindstorms NXT

Lego Mindstorms NXT [14] is essentially a programmable Lego toy. Due to its openness and the numerous ways to program, configure and use it, a community of technically interested people has grown around it further pushing the limits of usage. It is used at several universities, e.g. Lund University and RWTH Aachen, as a popular hardware platform in control engineering and mechatronic systems.

The heart of the unit is the NXT device (in the center of the image below).

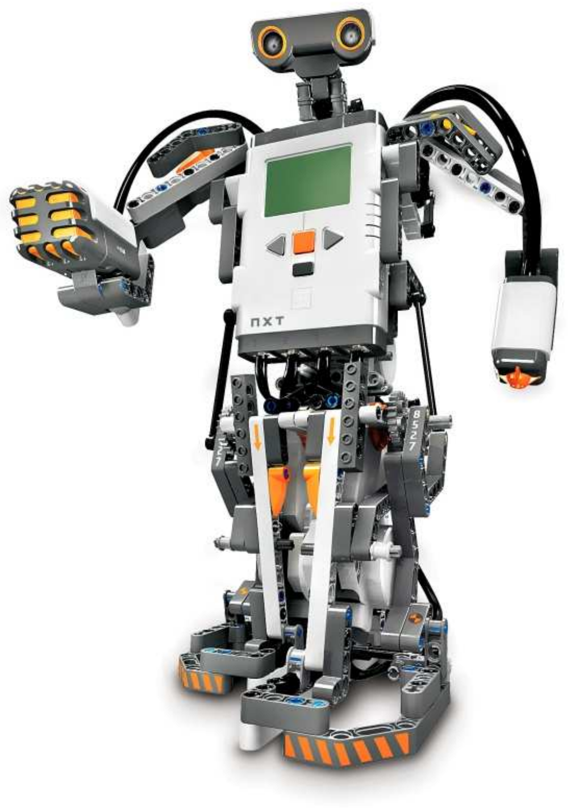


Figure 13. Lego Mindstorms robot.

The NXT device is a Lego brick containing a micro-controller with a graphical display, Bluetooth chip and interface (ports) to a number of sensors and actuators. It is configurable in numerous ways and the basic kit includes

- The NXT device
- Touch sensor
- Sound sensor
- Light sensor
- Ultrasonic sensor
- Servo motors
- Basic Lego building blocks.

In addition to the basic building blocks, additional add-on sensors and actuators can be acquired from third-party vendors.

HiTechnic

HiTechnic is a manufacturer of a big range of robotic sensors for the Lego Mindstorms NXT. Among their products are the following sensors:

- Acceleration / Tilt Sensor[1]
- Color Sensor
- Compass Sensor
- Gyro Sensor[1]
- IRReciver Sensor
- IRSeeker Sensor

Several of their sensors are available in the C API from nxtOSEK, presented in a coming section.

Mindsensors

Mindsensors is another manufacturer of accessories to the Lego Mindstorms NXT. Among their products are:

- Realtime Clock
- Multi-Sensitivity Acceleration Sensor[1]

Their products are not supported in the nxtOSEK C API. E.g. the Acceleration Sensor, is supported in NXT-G, NXC/NBC, RobotC but not in the C API from nxtOSEK. To use a Mindsensors sensor in the nxtOSEK Real-Time Operating System (RTOS), low level drivers had to be written, which make them a bit more hard-to-use than the HiTechnic sensors (for this particular platform). On the other hand, e.g. the Acceleration Sensor from Mindsensors is much more sensitive than the one from HiTechnic.

[1] Supported in LEGO_Mindstorms library implemented for this thesis

nxtOSEK

nxtOSEK, [15], is a freely available RTOS for Lego Mindstorms NXT. It is supplied as an open-source project and provides the user with, in particular:

- A programming environment using a GCC tool chain
- C API to Lego Mindstorms sensors and motors
- C API to some third-party sensors
- A large set of code examples

Based on the extensive set of examples supplied with nxtOSEK, a wrapper has been developed to form a base for the Dymola-generated fixed-point C code such that it can be compiled into an executable for the NXT using the standard GCC tool chain.

The nxtOSEK web page [15] also supplies several methods to download the compiled executable to the NXT device. The most trivial one uploads the program to an NXT without firmware installed. In that case the program is uploaded to and executed from the RAM memory of the device and it is then lost after termination of the device. Another alternative is to install an enhanced NXT firmware that allows you to upload the program to the flash memory of the device. The program is then kept in the memory when the NXT is restarted. When running the program it is copied to the RAM and executed from there just as for the first method. This alternative was found to be much more convenient than the first one, since the program is kept in the NXT after reboot. A third option is also described on the webpage [15] but it has not been tested during this work.

Cygwin

Cygwin [16] is a program that enables some Unix functionality on Windows by acting like a Unix-like environment.

In the scope of this thesis it is used as a command-line tool to compile and link the Dymola-generated code into an executable that can be run in the nxtOSEK RTOS. It is also used to download the executable code to the Lego device.

4.2 Tool chain

The tool chain when running the model on the Lego target is illustrated in Figure 14 below.

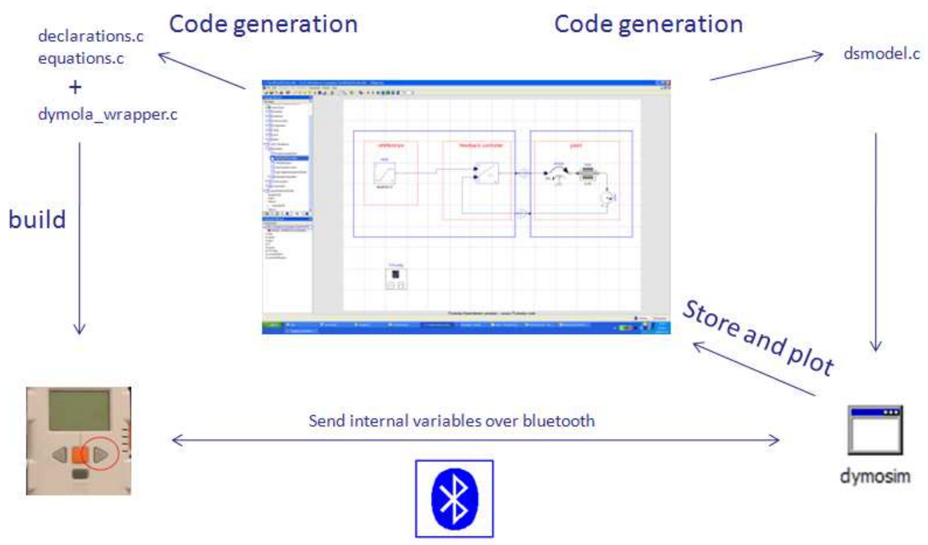


Figure 14. Tool chain.

Dymola is used to develop models of plant and controller to investigate effects of e.g. fixed-point aritmetics and sampling. Code can then be generated for the Lego target and using Cygwin the generated code is compiled and downloaded to the hardware. To evaluate the design and to collect data (data logging is described later in section 4.4) the Lego device can be run together with the simulation process from Dymola.

4.3 Dymola Lego Mindstorms API

The Dymola – Lego Mindstorms API consists of two main parts:

- LEGO_Mindstorms, a Modelica library with blocks that can be used to map signals to parts of the NXT's sensors and actuators
- dymola_wrapper.c, a framework/wrapper for the automatically generated model code from Dymola.

LEGO_Mindstorms library

The LEGO_Mindstorms library was developed to implement an API to a subset of the sensors and actuators available for the Lego Mindstorms NXT platform. The main structure of the library can be seen in Figure 15.

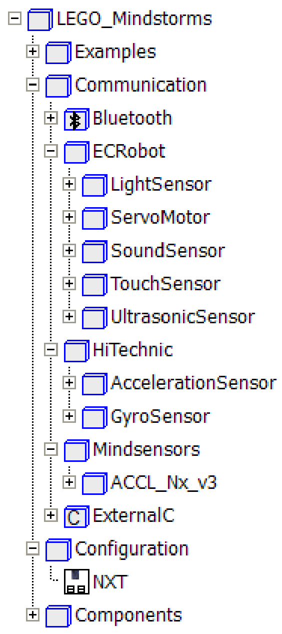
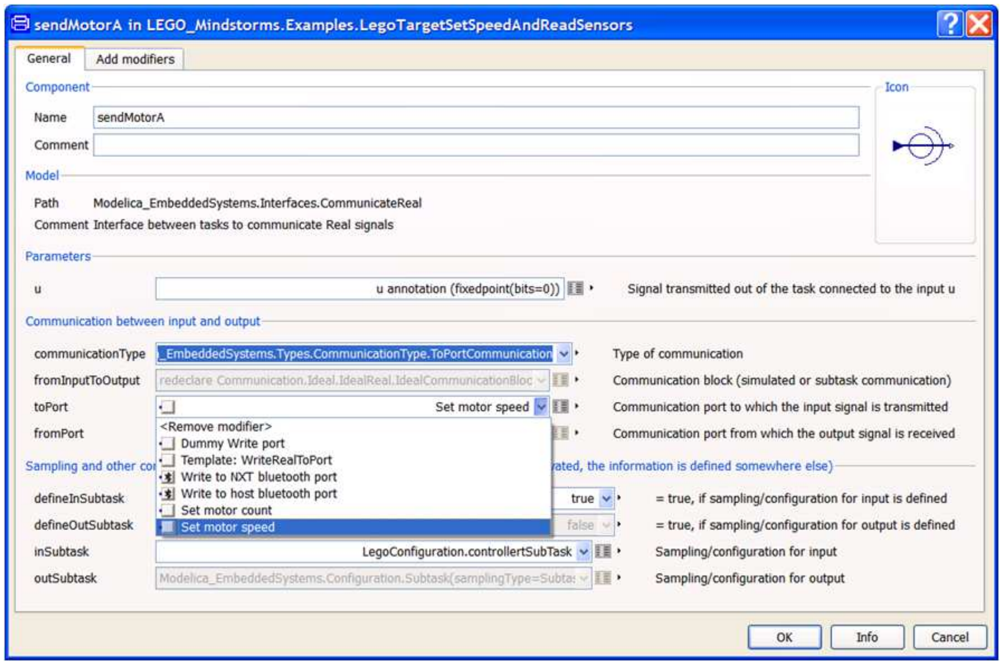


Figure 15. LEGO_Mindstorms Modelica library.

The library is designed to be compatible with Modelica_EmbeddedSystem in such a way that the API components can be accessible directly from the parameter dialog of the communicateReal block. This is done by extending from the architecture defined in Modelica_EmbeddedSystems, i.e., the components/blocks extend from a base class in Modelica_EmbeddedSystems.

The communicateReal can then be used in a model to map a signal to a specific low-level C function on the target. An example would be to send a control signal to a motor or to read from a sensor, e.g., touch sensor to detect contact.

The benefit of this design is that when the LEGO_Mindstroms library is loaded in Dymola, the new components will automatically appear as new choices that can be selected in the parameter dialog of communicateReal, shown in Figure 16, actuators, and in Figure 17, sensors.


sendMotorA in LEGO_Mindstorms.Examples.LegoTargetSetSpeedAndReadSensors

General | Add modifiers

Component
Name: sendMotorA
Comment:

Model
Path: Modelica_EmbeddedSystems.Interfaces.CommunicateReal
Comment: Interface between tasks to communicate Real signals

Parameters
u: u annotation (fixedpoint(bits=0)) — Signal transmitted out of the task connected to the input u

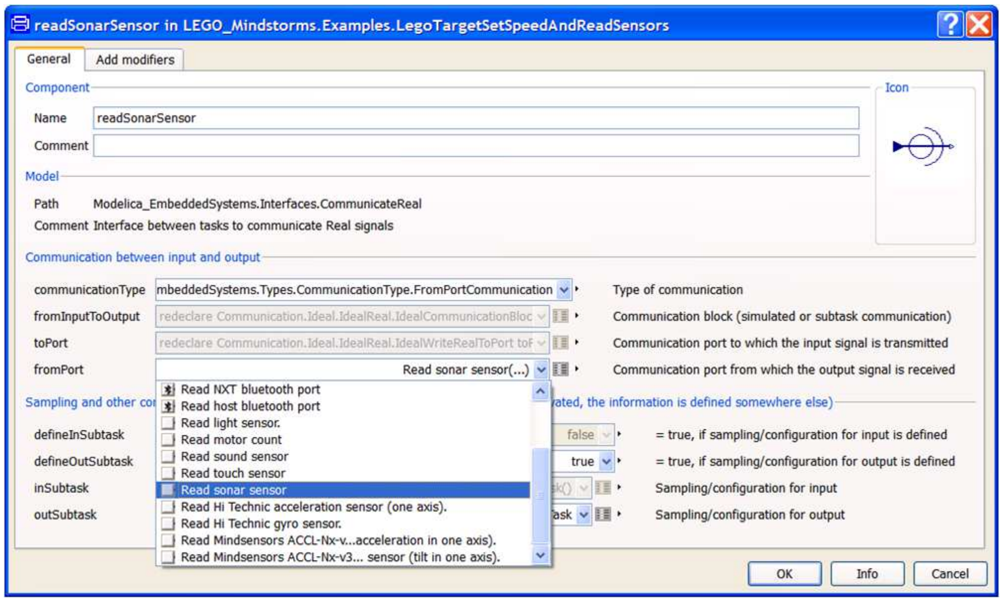
Communication between input and output
communicationType: _EmbeddedSystems.Types.CommunicationType.ToPortCommunication — Type of communication
fromInputToOutput: redeclare Communication.Ideal.IdealReal.IdealCommunicationBloc — Communication block (simulated or subtask communication)
toPort: Set motor speed — Communication port to which the input signal is transmitted
fromPort: — Communication port from which the output signal is received

<Remove modifier>
Dummy Write port
Template: WriteRealToPort
Write to NXT bluetooth port
Write to host bluetooth port
Set motor count
Set motor speed

Sampling and other co... ated, the information is defined somewhere else)
defineInSubtask: true — = true, if sampling/configuration for input is defined
defineOutSubtask: false — = true, if sampling/configuration for output is defined
inSubtask: LegoConfiguration.controllertSubTask — Sampling/configuration for input
outSubtask: Modelica_EmbeddedSystems.Configuration.Subtask(samplingType=Subta — Sampling/configuration for output

OK | Info | Cancel

Figure 16. NXT actuators in the communicateReal parameter dialog


readSonarSensor in LEGO_Mindstorms.Examples.LegoTargetSetSpeedAndReadSensors

General | Add modifiers

Component
Name: readSonarSensor
Comment:

Model
Path: Modelica_EmbeddedSystems.Interfaces.CommunicateReal
Comment: Interface between tasks to communicate Real signals

Communication between input and output
communicationType: mbeddedSystems.Types.CommunicationType.FromPortCommunication — Type of communication
fromInputToOutput: redeclare Communication.Ideal.IdealReal.IdealCommunicationBloc — Communication block (simulated or subtask communication)
toPort: redeclare Communication.Ideal.IdealReal.IdealWriteRealToPort toF — Communication port to which the input signal is transmitted
fromPort: Read sonar sensor(...) — Communication port from which the output signal is received

Read NXT bluetooth port
Read host bluetooth port
Read light sensor.
Read motor count
Read sound sensor
Read touch sensor
Read sonar sensor
Read Hi Technic acceleration sensor (one axis).
Read Hi Technic gyro sensor.
Read Mindsensors ACCL-Nx-v...acceleration in one axis).
Read Mindsensors ACCL-Nx-v3... sensor (tilt in one axis).

Sampling and other co... ated, the information is defined somewhere else)
defineInSubtask: false — = true, if sampling/configuration for input is defined
defineOutSubtask: true — = true, if sampling/configuration for output is defined
inSubtask: — Sampling/configuration for input
outSubtask: — Sampling/configuration for output

OK | Info | Cancel

Figure 17. NXT sensors in the communicateReal parameter dialog

If additional sensors or actuators are needed, a user can follow the same design principle and implement his own API block that also will appear automatically in the dialogs. This makes the framework very flexible for the user.

The current implementation supports a subset of the nxtOSEK C API

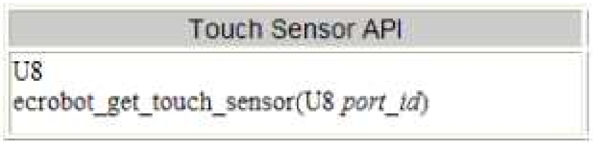
- ECRobot
  - Servo motor
  - Light sensor

    - Sound sensor
    - Touch sensor
    - Ultrasonic sensor
- HiTechnic
    - Acceleration sensor (NAC1040)
    - Gyro sensor (NGY1044)
- Mindsensors
    - Acceleration sensor (ACCL-Nx-v3)

described in more detail below.

NXT standard I/O modules

The ECRobot sub-package contains an interface to the standard Lego Mindstorms sensors and actuators. Each block contains a mapping to the corresponding nxtOSEK C API function, adding min/max values whenever possible. The blocks extends from a base class in Modelica_EmbeddedSystems and then uses the Modelica external function concept to map the signal to the C function from the API. As an example consider the touch sensor. The API, as can be seen in Figure 18, uses the U8 type (unsigned 8-bit integer) as input and return value.



Touch Sensor API
U8 ecrobot_get_touch_sensor(U8 *port_id*)

Figure 18. nxtOSEK Touch Sensor API

The corresponding Modelica function in LEGO_Mindstorms is

```
function ecrobot_get_touch_sensor
  input Integer port_id(min=0, max=3)
  "0 = NXT_PORT_S1, 1 = NXT_PORT_S2, 2 = NXT_PORT_S3, 3 = NXT_PORT_S4";
  input Real Time;
  output Real signal;
external "C" signal = ecrobot_get_touch_sensor(port_id);
end ecrobot_get_touch_sensor;
```

As can be seen above, the Modelica function uses the Modelica external function concept (external “C” y =foo(u)) and contains the call to the actual C function as defined in the API. The input variable *Time* was needed make the function time varying since Dymola otherwise evaluates the function. For the above function to fit in the Modelica_EmbeddedSystems framework, i.e, to make it selectable in the communicateReal block as in Figure 17, a Modelica block has been constructed by extending the appropriate base class. This block, as shown below, contain a call to the function above as well as assigning min/max attributes of the output of the sensor.

```
block ecrobot_get_touch_sensor "Read touch sensor"
  extends Modelica_EmbeddedSystems.Interfaces.BaseReal.
    PartialReadRealFromPort(minValue=0, maxValue=1,
      y(min=0, max=1));
  parameter Integer port_id(min=0, max=3) = 0
  "0 = NXT_PORT_S1, 1 = NXT_PORT_S2, 2 = NXT_PORT_S3, 3 = NXT_PORT_S4";
equation
  // Returns touch sensor status. 0 = not touched, 1 = touched.
  y = ExternalC.ECRobot.TouchSensor.ecrobot_get_touch_sensor(port_id, time);
end ecrobot_get_touch_sensor;
```

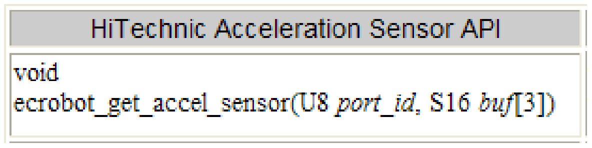
Note that the Modelica implementation returns a Real whereas the C function returns an integer (U8). The reason for this is to make all sensors and actuators available from the communicateReal block (communicateReal handles real signals as the name indicates). This will make usage straight-forward, the user can always use communicateReal and know that all supported components can be found there. If the formally more correct approach was used, i.e., using communicateInteger and communicateBoolean as well, then the user would have to use the C API to figure out what return type the underlying C function has and then use the appropriate communication block type.

HiTechnic Gyro Sensor

The HiTechnic Gyro Sensor is a third party sensor form HiTechnic and is found in the HiTechnic sub-package. The implementation was straight-forward, similar to the one described in NXT standard I/O modules above (nxtOSEK contains a C API for this sensor in the same style as above).

HiTechnic Acceleration Sensor

The Acceleration Sensor, found in the HiTechnic sub-package, also had a C API available in nxtOSEK as can be seen below in Figure 19.



HiTechnic Acceleration Sensor API
void ecrobot_get_accel_sensor(U8 *port_id*, S16 *buf*[3])

Figure 19. nxtOSEK Acceleration sensor API

It is based on passing pointers but the communication blocks do not take pointers (arrays) only scalars so a C wrapper was developed to interface it. The idea is simple. In Dymola you select which axis you would like to use with a parameter and then that axis is extracted from the array in the C wrapper and passed as a scalar variable to the Modelica block. The C implementation is

```
/* Wrapper to read one axis from the Hi Technic Acceleration
sensor */
S16 get_accel_axis(U8 port_id, U8 axis)
{
	S16 buffer[3];
	ecrobot_get_accel_sensor(port_id, buffer);
	return buffer[axis];
}
```

and the corresponding Modelica block

```
block get_accel_axis
  "Read Hi Technic acceleration sensor (one axis)."
  extends Modelica_EmbeddedSystems.Interfaces.BaseReal.
```

```
    PartialReadRealFromPort(y(min=-600, max=600),
      minValue=-600, maxValue=600);
  parameter Integer port_id(
    min=0,
    max=3) = 0
  "0 = NXT_PORT_S1, 1 = NXT_PORT_S2, 2 = NXT_PORT_S3, 3 = NXT_PORT_S4";
  parameter Integer axis(min=0,max=2)=0 "0 = X-axis, 1 = Y-axis, 2 = Z-axis";
equation
  // Returns raw A/D data from one axis from the accel sensor
  y = LEGO_Mindstorms.Communication.ExternalC.HiTechnic.
  AccelerationSensor.get_accel_axis (port_id, axis, time);
end get_accel_axis;
```

Mindsensors

In the Mindsensors sub-package another, third-party acceleration sensor is found; the ACCL-Nx-v3 acceleration sensor, which is more sensitive than the acceleration sensor from HiTechnic. This sensor is not supported in the nxtOSEK C API so both the Modelica part and the C part had to be implemented. The C implementation is based on reading raw data from the I2C bus, on which the sensor is connected, and then use wrappers to extract either tilt or acceleration in the selected axis. Below the function for reading raw data and the wrappers.

```
S16 mindsensors_get_accel_axis(U8 port_id, U8 axis)
{
      return mindsensors_get_accel_sensor(port_id, axis+3);
}

S16 mindsensors_get_tilt_axis(U8 port_id, U8 axis)
{
      return mindsensors_get_accel_sensor(port_id, axis);
}

S16 mindsensors_get_accel_sensor(U8 port_id, U8 axis)
{
      static S16 state[6];
      static U8 data[9];
      if (i2c_busy(port_id) == 0)
      {
            /* tilt data */
            state[0] = (S16)data[0];
            state[1] = (S16)data[1];
            state[2] = (S16)data[2];
            /* 10 bit acceleration data */
            state[3] = (S16)data[3] + ((S16)data[4] << 8);
            state[4] = (S16)data[5] + ((S16)data[6] << 8);
            state[5] = (S16)data[7] + ((S16)data[8] << 8);
            i2c_start_transaction(port_id,1,0x42,1,&data[0],9,0);
      }
      return state[axis];
}
```

The corresponding Modelica blocks are implemented just as for the previous examples.

Bluetooth

The Bluetooth sub-package contains Modelica blocks to access Bluetooth communication routines from within the model. An example of usage is to send

reference signals from the host computer to the Lego Mindstorms NXT device. The implementation has been made such that the user can use the communication routines in the same convenient way as the previously described sensors and actuators, just by selecting from the pull-down menu of the communicateReal block, see e.g. Figure 16 and Figure 17.
The available blocks are

- nxt_read_bluetooth
- nxt_write_bluetooth
- host_read_bluetooth
- host_write_bluetooth

The blocks call the underlying Modelica functions needed for the Dymola Bluetooth communication API (host_…_bluetooth) developed for this thesis, as well as Modelica functions for the Bluetooth communication routines in the nxtOSEK C API (nxt_..._bluetooth).

In order to use the Bluetooth communication features, a communication channel had to be established between the host computer and the Lego Mindstorms NXT device. To make this setup as convenient as possible for the user a component that opens and closes a channel was implemented. The component can be found under the sub-package Components and has the icon depicted below and contains the following Modelica code:

```
block Bluetooth
  parameter Integer port=8 "Virtual COM port number";
initial algorithm
  LEGO_Mindstorms.Communication.ExternalC.BlueTooth.openChannel(port);
equation
  when terminal() then
    LEGO_Mindstorms.Communication.ExternalC.BlueTooth.closeChannel();
  end when;
end Bluetooth;
```

The code ensures that the communication channel is opened before any computations are made and that the channel is properly closed at the end of the simulation. The input parameter “port” is the numeric value of the assigned virtual COM port. This value is determined when the device is paired with the host computer, using, e.g., Windows (the device must be paired and connected before any communication features can be used).

dymola_wrapper

This section describes the dymola_wrapper.c framework that is used as a base for the automatically generated model code in order to make it run on the Lego Mindstorms NXT under the nxtOSEK operating system. The code is based on examples from the nxtOSEK source code and is outlined below.

```
<includes>
```

```
#include "target_port.h"

/* OSEK declarations */

/* Include fixedpoint variable declarations */
#include "declarations.c"
/* Include API to sensors from Mindsensors */
#include "mindsensors.c"
int startTime = 0;

/* LEJOS OSEK hooks */
 <code>
/* LEJOS OSEK hook to be invoked from an ISR in
category 2 */
 <code>
/* Wrapper to read one axis from the Hi Technic
Acceleration sensor */
S16 get_accel_axis(U8 port_id, U8 axis)
{
 <code>
}

/* Task1 executed every x msec */
TASK(Task1)
{
     /* map system time to fixedpoint time */
     /* reset motor count to 0 */
     <code>

     /* include fixedpoint equations */
     #include "equations.c"

     /* display time in seconds*/
     <code>

     TerminateTask();
}
```

The essential part is the task (Task1) that is executed periodically. It contains (by an include statement) the model equations as generated by the fixed-point machinery of Dymola. Every time the task is executed the model equations are recomputed with updated inputs and sensor values.

A detail that is not obvious at first glance is the first couple of lines in the task starting with the comments “ /* map system time …”. The full code is

```
     /* map system time to fixedpoint time */
     /* reset motor count to 0 */
     if (startTime == 0) {
          // only executed the first execution cycle
          startTime=systick_get_ms();
          nxt_motor_set_count(NXT_PORT_B,0);
```

```
            nxt_motor_set_count(NXT_PORT_C,0);
        }
        time_FP = (int)1024*(systick_get_ms()-
startTime)/1000;
```

The purpose of this code is to scale the time variable for, e.g., time-dependent reference signals such as ramps and to remove bias. The time variable is constructed by using a built in millisecond counter. The counter is started when the Lego Mindstorms NXT device is powered up and will thus always be biased. In the Dymola-generated model equations, the time variable is by default scaled with 10 fractional bits and thus we need to incorporate that scaling (1024) when updating the time variable as well as rescale it from milliseconds to seconds. To remove the bias, the value of the counter at the first execution is stored and then subtracted each update. At first execution the counters of the motors (in the example above connected to port B and C) are also reset to make sure they always start counting on 0 when the code starts.

The last section is an example how to output variables to the Lego Mindstorms NXT display for, e.g., debugging. Below is example code to display the time variable (in seconds) on the display, both scaled and uscaled (raw).

```
        /* display time in seconds*/
        display_clear(0);
        display_goto_xy(0, 0);
        display_string("My display");
        display_goto_xy(0, 2);
        display_string("TIME:");
        display_int(time_FP/1024, 0);
        display_goto_xy(0,4);
        display_string("TIME unscaled:");
        display_int(time_FP, 0);
        display_update();
```

4.4 Dymola Bluetooth interface for plotting and animation

Debugging embedded systems can be a very difficult and time-consuming task. One of the main problems is that it is usually very hard to get usable/reliable data of the internal state of the embedded system, only a few inputs/outputs are available. The reason for this is that many embedded systems lack an internal file system or if one exist, the area of persistent storage would likely be relatively small and would thus quickly fill if one attempts to use it for data logging.

To facilitate automatic data logging for the Lego Mindstorms NXT device, we utilize the fact that it has a built-in Bluetooth chip and, using nxtOSEK, a C API for read/write operations. The user can enable this feature by setting a flag in the Dymola command prompt and if activated, code will automatically be generated to support data logging of the internal signals.

The basic idea is to generate two variants of the model code, one (in fixed-point) to be downloaded to the embedded system and one for the PC with Dymola that instead of computing values listens on a Bluetooth communication channel and stores the information received from the embedded system. The code for the embedded system (Lego Mindstorms NXT) contains, in addition to the model equations in fixed-point, also Bluetooth send commands.

In the current implementation, there is no way to select which variable shall be logged. If the flag is set then all variables are logged and if not set none are logged. During testing, approximately 30 signals could be logged while running the system with 10ms sample rate. This implicitly puts a limit on the size of the model when the automatic data logging can be used. If this limit is exceeded then plotting and animation cannot be run.

4.5 Dymola Bluetooth interface for direct communication

Above we discussed automatic data logging using the Bluetooth communication channel. The Bluetooth channel cannot only be used for that purpose, it can also be used for direct communication with the model from the PC with Dymola. An example, that has been implemented and tested, is to send reference values to a controller from a model in Dymola.

Using the Modelica_EmbeddedSystems framework, Bluetooth communication blocks have been implemented and can be reached in the fromPort and toPort modes of the communication points as can be seen in Figure 3, section 2.1. The Modelica implementation is a mapping to the actual C routines that is used for the communication between target and host. The target routines (on the Lego Mindstorms NXT) make use of the nxtOSEK Bluetooth C API.

The tested example used a USB game controller connected to Dymola to generate reference speed signals for the wheels of the Lego Mindstorms NXT. Those signals where then sent to the NXT using the Bluetooth components described in section 4.3.

4.6 Online plotting and animation

Online plotting and animation is closely coupled to Bluetooth data logging when running the code on the Lego Mindstorms NXT. Without it, there would be no data to plot or animate. We thus need to separate the two cases, internal target (fixed-point simulation in Dymola) and external target (Lego Mindstorms NXT).

Internal fixed-point target

With internal target we denote running a Software-In-the-Loop simulation in Dymola to investigate the effects of fixed-point arithmetics on, e.g., controller performance. The typical scenario is to have a system with a plant and a controller, decoupled using components from Modelica_EmbeddedSystems. With the support for fixed-point activated, code is generated to compute both the normal floating-point computations of the model equations and their fixed-point counter parts. The user can for example run the model with the fixed-point equations as slaves to the real controller, acting as “fixed-point sampling” of the signals, as well as driving the system with the fixed-point controller. A parameter is introduced in the variable browser to toggle between the two modes without the need to re-translate the model.

Since the code always computes both the floating-point and fixed-point versions of a variable, the signals can be compared for analysis. Under a virtual fixed-point component in the Dymola variables browser (as well as in the .mat result file) all variables that are computed using fixed-point have two clones, see Figure 20;

- <Name>_fromfixedpoint
- <Name>_original

The “original” signal is the signal computed using floating-point and the “fromfixedpoint”, as the name indicates, is the recovered value when rescaling the fixed-point value back to a real value.

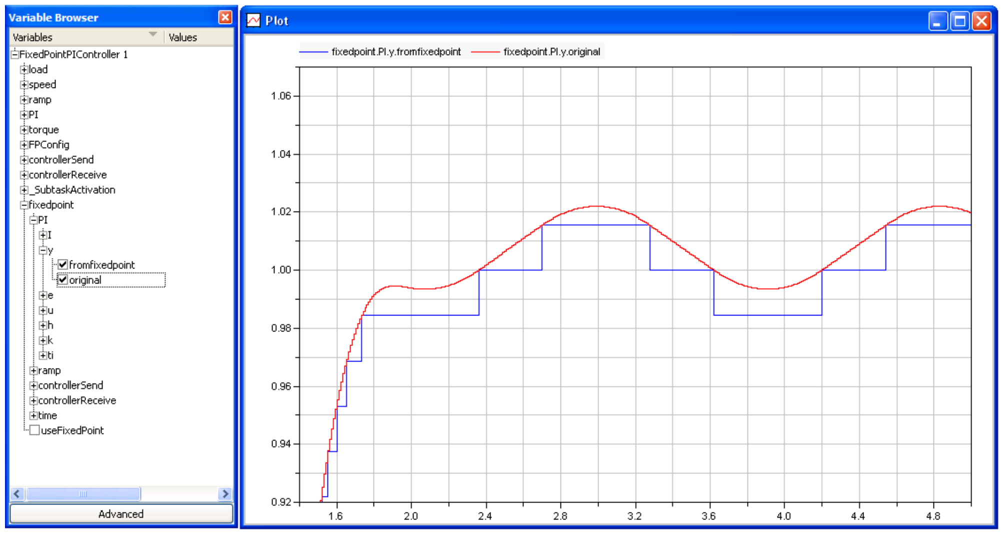


Figure 20. Variable browser and fixed-point plot

The plotting itself is the same as when running any other model in Dymola, online in the sense that values are reloaded in the plot as often as the user has specified in the experiment setup.

External target

When running the code on an external target, online plotting (and animation) depends on the use of Bluetooth data logging. The basic idea differs a bit from plotting and logging when running on an internal target.

The basic idea is to be able to plot signals for debugging purposes. Only using the recovered values one can be fooled since they are automatically rescaled correctly. Particularly when using sensors and actuators it is preferable to be able to see the raw data as seen by the hardware. The structure in the variable browser is the same but the interpretation of “original” and “fromfixedpoint” differs. Just as before, “fromfixedpoint” is the recovered value, comparable with any other signal since it is rescaled to a real value. The “original” value is now the raw integer data as seen by the hardware. The original values are not comparable to each other, since they can vary very much in magnitude due to their scaling. But as mentioned, sometimes it can be critical to be able to see what the raw value is.

4.7 Lego Segway

The fixed-point capability of Dymola described in this thesis has been used in teaching at the Department of Automatic Control at Lund University in the advanced level course FRT090 – *Projects in Automatic Control*. Two groups selected the Dymola project in 2009, one group in the spring of 2010, two groups in the spring of 2011 and two groups in the spring of 2012.

In the projects, Dymola was used to model a Lego Segway, and to design a stabilizing controller, [17]. Fixed-point code for the controller was then generated and downloaded to the Lego target. Using the Bluetooth interface, data could be collected for plotting and animation, see Figure 21.

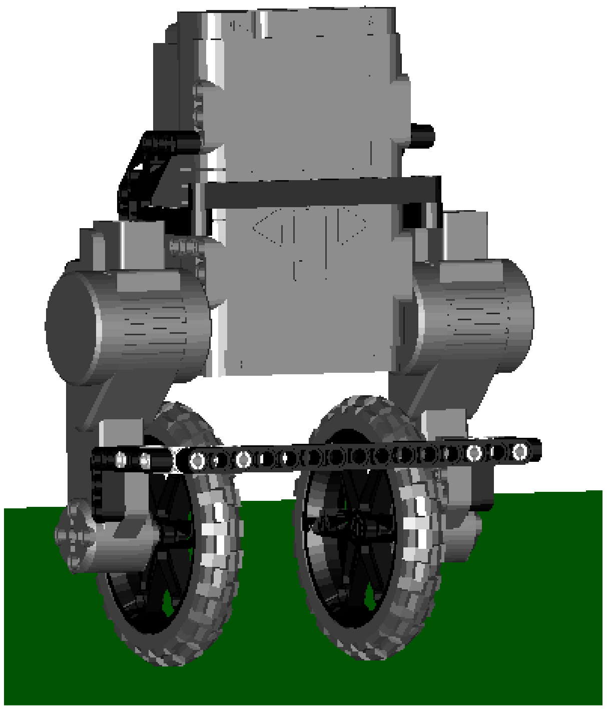


Figure 21. Dymola animation of the Lego Segway

4.8 Elektor Wheelie

During 2011 there was a Master's Thesis project, *Modeling, Control and Automatic Code Generation for a Two-Wheeled Self-Balancing Vehicle Using Modelica* [18], that used the features described in this thesis for control of a full-scale Segway [19] clone, ElektorWheelie [20], capable of carrying a person.

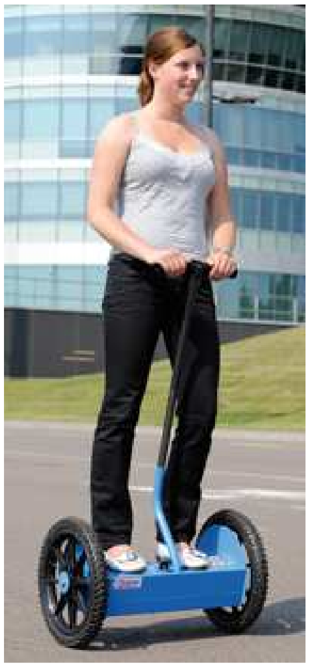


Figure 22. Elektor Wheelie in action

Using Modelica_EmbeddedSystems to partition the system model, Figure 23, and the automatic fixed-point code generation capabilities of Dymola, controller code was generated and downloaded to the ElektorWheelie. The students made several experiments and compared the Dymola generated code with manually written fixed-point code and concluded:

- "The results were satisfactory from an experimental point of view, the estimators and controller achieve the control objective and it was verified that the automatic code generation by Dymola manages to be as accurate as the manual fixed-point coding."
- "The manual and automatically generated code performance was tested during experimental rides. There was no significant difference between both results which shows that the automatic code generation is a useful tool comparable to the manual coding"

The report also mentions some areas of improvements. As an example the Atmega32 processor of the ElektorWheelie does not support division in its instruction set. The code had to be manually adapted to convert division by a power of two to left shifts. However, testing on other platforms indicated that modern compilers can handle this automatically.

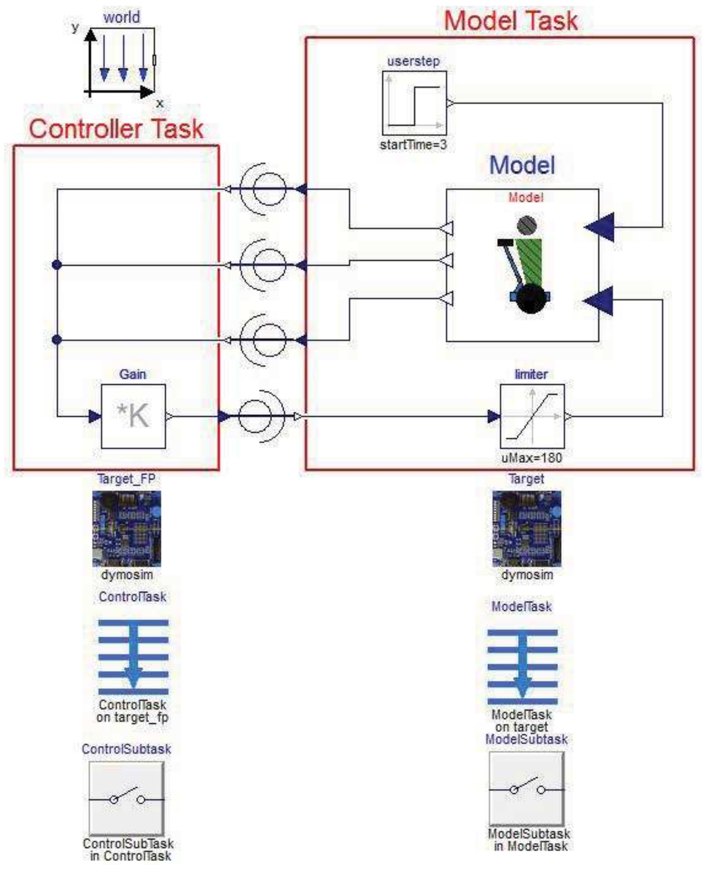


Figure 23. Modelica system model of the ElektorWheelie and controller.

5. Summary

5.1 Results and conclusions

The work in this thesis has in addition to this report also resulted in two conference papers, [1] and [21]. It has also been used in project courses and a Master's Thesis in automatic control at Lund University. Although it is far from a product ready for release, the main principle and strategy has been demonstrated to work in the above mentioned projects.

Experimental Modelica annotations were introduced to complement the existing attributes of Modelica variables to be able to set resolution needed for fixed-point. Two annotations were introduced to conveniently input the resolution with either an absolute or relative measure. It would be even more convenient if resolution existed as an attribute, since then there would be no need to use annotations as modifiers which would give more compact and easy-to-read Modelica code.

Using the new experimental annotations, Dymola was extended with functionality for analysis and code generation for fixed-point. Two different methods of range analysis were implemented; bit propagation and interval analysis. Interval analysis, although conservative, was concluded to never give larger intervals than bit propagation and was selected as the active method. It would be desirable to further analyze the expressions and improve that range analysis to give tighter intervals when possible.

The scaling of variables is based on user input (using the experimental annotations) as well as heuristics and some rules to propagate bits through the expressions. With a deeper analysis it should be possible to find "smarter" scaling to guarantee that no overflow can occur while minimizing the precision loss.

The code generation was intended to be portable but was influenced by the Lego Mindstorms target since that was the only platform we tested on ourselves. As an example it was detected in the Master's Thesis [17] that the generated divisions by a power of two are not supported on the ElektorWheelie processor. More user configurability would be desirable but it would also increase the complexity of the code.

The implementation of Bluetooth communication for data logging turned out to be a very good complement to "on-screen-debugging" on the Lego Mindstorms device. Less optimal was the fact that in this first implementation only one task is supported. This meant, e.g., that the controller code and code for data logging resided in the same task and all calculations were executed with the same priority. More desirable would have been to generate the data logging code in a separate task that could run with lower priority to avoid it influencing control performance. The same holds for reference signal generation using, e.g., a gamepad. Ideally that code should also be run in a separate task with lower priority.

Using the Lego Mindstorms device, a full Modelica-model to embedded code scenario could be tested and evaluated which was one of the original goals set for the thesis.

5.2 Future work

Some interesting topics for future work are:

- Range analysis of nonlinear functions and user-written functions by offline evaluation based on the range of the inputs.

- More sophisticated fixed-point analysis enabling smarter bit shifting for addition, subtraction, multiplication and division to reduce losses in accuracy.

- Generate tables with interpolation to support functions in fixed-point with a user-specified resolution and range. Identify periodic functions.

- Introduce guard bits for variables with uncertain ranges.

- Asserts for overflow in the fixed-point code.

- User specified rounding functions.

- More advanced range analysis.

Bibliography

[1] Ulf Nordström, José Díaz López, and Hilding Elmqvist, *Automatic Fixed-point Code Generation for Modelica using Dymola.* Conference Proceedings of the International Modelica Conference, Vienna, Austria, 2006.

[2] Multi-Engineering Modeling and Simulation - Dymola - CATIA - Dassault Systèmes, April 2012. URL `http://www.3ds.com/products/catia/portfolio/dymola`

[3] Peter Fritzson, P*rinciples of Object-Oriented Modeling and Simulation with Modelica 2.1.* IEEE PRESS, 2004.

[4] Michael M. Tiller, *Introduction to Physical Modeling with Modelica*. Kluwer Academic Publishers, 2001.

[5] Modelica and the Modelica Association, April 2012. URL `http://www.modelica.org`

[6] Hilding Elmqvist, Martin Otter, Dan Henriksson, Bernhard Thiele, and Sven Erik Mattsson, *Modelica for Embedded Systems.* Conference Proceedings of the International Modelica Conference, Como, Italy, 2009. DOI: 10.3384/ecp09430096.

[7] *Modelica - A Unified Object-Oriented Language for Physical Systems Modeling, Language Specification, Version 3.1.* Modelica Association, 2009.

[8] Two's complement - Wikipedia, April 2012. URL `http://en.wikipedia.org/wiki/Two's_complement`

[9] Texas Instruments - TMS320C64x DSP Library, October 2003. URL `http://focus.ti.com/lit/ug/spru565b.pdf`

[10] International Organization for Standardization, *Programming languages - C. Draft Internaltional Standard ISO/IEC DIS 9899,* 1989.

[11] Randy Yates, *Fixed-Point Arithmetic: An Introduction.* Digital Signal Labs, 2009.

[12] R. E. Moore, *Interval Analysis.* Prentice-Hall, 1966. ISBN 0-13-476853-1.

[13] Interval arithmetic - Wikipedia, April 2012. URL `http://en.wikipedia.org/wiki/Interval_arithmetic`

[14] Lego.com MINDSTORMS : Home, April 2012. URL `http://mindstorms.lego.com`

[15] nxtOSEK : index, April 2012. URL `http://lejos-osek.sourceforge.net`

[16] Cygwin, April 2012. URL `http://www.cygwin.com`

[17] Joel Pettersson, Pär Isaksson, Sofia Dahlberg, Stefan Flixeder. *Lego Robot with Modelica/Dymola*. Project report in FRT090, Department of Automatic Control, Lund University, 2011.

[18] Carabel, Carlos Javier Pedreira och García, Andrés Alejandro Zambrano, *Modeling, Control and Automatic Code Generation for a Two-Wheeled Self-Balancing Vehicle Using Modelica.* Master Thesis, Department of Automatic Control, Lund University, 2011. ISRN LUTFD2/TFRT--5884--SE.

[19] Segway – The leader in personal, green transportation, April 2012. URL `http://www.segway.com/`

[20] ElektorWheelie, April 2012. URL `https://www.elektor.com/projects/elektorwheelie.986808.lynkx`

[21] Johan Åkesson, Ulf Nordström, and Hilding Elmqvist, *Dymola and Modelica_EmbeddedSystems in Teaching - Experiences from a Project Course.* Conference Proceedings of the International Modelica Conference, Como, Italy, 2009. DOI: 10.2284/ecp09430086.